Oracle9*i*

SQL Reference

Release 2 (9.2)

March 2002

Part No. A96540-01

ORACLE®

Oracle9*i* SQL Reference, Release 2 (9.2)

Part No. A96540-01

Primary Author: Diana Lorentz

Contributing Author: Joan Gregoire

Contributors: Sundeep Abraham, Nipun Agarwal, Dave Alpern, Angela Amor, Patrick Amor, Rick Anderson, Vikas Arora, Lance Ashdown, Hermann Baer, Subhransu Basu, Ruth Baylis, Paula Bingham, Rae Burns, Yujie Cao, Larry Carpenter, Sivasankaran Chandrasekar, Thomas Chang, Tim Chorma, Lex de Haan, Norbert Debes, George Eadon, Bill Gietz, Ray Guzman, John Haydu, Lilian Hobbs, Jiansheng Huang, Ken Jacobs, Archna Johnson, Vishy Karra, Thomas Keefe, Susan Kotsovolos, Muralidhar Krishnaprasad, Goutam Kulkarni, Paul Lane, Shilpa Lawande, Geoff Lee, Yunrui Li, Lenore Luscher, Kevin MacDowell, Anand Manikutty, Vineet Marwah, Steve McGee, Bill McGuirk, Bill McKenna, Meghna Mehta, Tony Morales, Sujatha Muthulingam, Michael Orlowski, Jennifer Polk, Dmitry Potapov, Rebecca Reitmeyer, Kathy Rich, John Russell, Vivian Schupmann, Shrikanth Shankar, Vikram Shukla, Mike Stewart, Sankar Subramanian, Seema Sundara, Hal Takahara, Ashish Thusoo, Anh-Tuan Tran, Randy Urbano, Guhan Viswanathan, David Wang, Jim Warner, Andy Witkowski, Daniel Wong, Jianping Yang, Adiel Yoaz, Qin Yu, Tim Yu, Mohamed Zait, Fred Zemke

Contents

7 Common SQL DDL Clauses

8 SQL Queries and Subqueries

9 SQL Statements: ALTER CLUSTER to ALTER SEQUENCE

10 SQL Statements: ALTER SESSION to ALTER SYSTEM

11 SQL Statements: ALTER TABLE to ALTER TABLESPACE

12 SQL Statements: ALTER TRIGGER to COMMIT

13 SQL Statements: CREATE CLUSTER to CREATE JAVA

14 SQL Statements: CREATE LIBRARY to CREATE SPFILE

15 SQL Statements: CREATE SYNONYM to CREATE TRIGGER

16 SQL Statements: CREATE TYPE to DROP ROLLBACK SEGMENT

17 SQL Statements: DROP SEQUENCE to ROLLBACK

Index

Send Us Your Comments

Oracle9*i* SQL Reference, Release 2 (9.2)

Part No. A96540-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

This reference contains a complete description of the Structured Query Language (SQL) used to manage information in an Oracle database. Oracle SQL is a superset of the American National Standards Institute (ANSI) and the International Standards Organization (ISO) SQL99 standard.

This preface contains these topics:

Audience

The *Oracle9i* SQL Reference is intended for all users of Oracle SQL.

Organization

This reference is divided into the following parts:

Volume 1

Chapter 1, "Introduction"

This chapter discusses the history of SQL and describes the advantages of using it to access relational databases.

Chapter 2, "Basic Elements of Oracle SQL"

This chapter describes the basic building blocks of an Oracle database and of Oracle SQL.

Chapter 3, "Operators"

This chapter describes SQL operators.

Chapter 4, "Expressions"

This chapter describes SQL expressions.

Chapter 5, "Conditions"

This chapter describes SQL conditions.

Chapter 6, "Functions"

This chapter describes how to use SQL functions.

Chapter 7, "Common SQL DDL Clauses"

This chapter describes a number of DDL clauses that are frequently used in multiple top-level SQL statements.

Chapter 8, "SQL Queries and Subqueries"

This chapter describes the different types of SQL queries and lists the various types of SQL statements.

Volume 2

Chapter 9, "SQL Statements: ALTER CLUSTER to ALTER SEQUENCE"
Chapter 10, "SQL Statements: ALTER SESSION to ALTER SYSTEM"
Chapter 11, "SQL Statements: ALTER TABLE to ALTER TABLESPACE"
Chapter 12, "SQL Statements: ALTER TRIGGER to COMMIT"
Chapter 13, "SQL Statements: CREATE CLUSTER to CREATE JAVA"
Chapter 14, "SQL Statements: CREATE LIBRARY to CREATE SPFILE"
Chapter 15, "SQL Statements: CREATE SYNONYM to CREATE TRIGGER"
Chapter 16, "SQL Statements: CREATE TYPE to DROP ROLLBACK SEGMENT"
Chapter 17, "SQL Statements: DROP SEQUENCE to ROLLBACK"
Chapter 18, "SQL Statements: SAVEPOINT to UPDATE"
These chapters list and describe all Oracle SQL statements in alphabetical order.

Appendix A, "How to Read Syntax Diagrams"
This appendix describes how to read the syntax diagrams in this reference.

Appendix B, "Oracle and Standard SQL"
This appendix describes Oracle compliance with ANSI and ISO standards.

Appendix C, "Oracle Reserved Words"
This appendix lists words that are reserved for internal use by Oracle.

Appendix D, "Examples"
This appendix provides extended examples that use multiple SQL statements and are therefore not appropriate for any single section of the reference.

Structural Changes in the SQL Reference in Oracle9*i* Release 2 (9.2)

The following frequently used DDL clauses have been separated into their own chapter, Chapter 7, "Common SQL DDL Clauses": *allocate_extent_clause* on page 7-2, *constraints* on page 7-5, *deallocate_unused_clause* on page 7-37, *file_specification* on page 7-39, *logging_clause* on page 7-45, *parallel_clause* on page 7-49, *physical_attributes_clause* on page 7-52, *storage_clause* on page 7-56.

In earlier releases, the *autoextend_clause* appeared in a number of SQL statements. It now is documented as part of the *datafile_tempfile_spec* form

of *file_specification*, to clarify that this attribute relates to datafiles and tempfiles.

Structural Changes in the SQL Reference in Oracle9*i* Release 1 (9.0.1)

The chapter that formerly described expressions, conditions, and queries has been divided. Conditions and expressions are now two separate chapters, and queries are described in Chapter 8, "SQL Queries and Subqueries".

CAST, DECODE, and EXTRACT (datetime), which were formerly documented as forms of expression, are now documented as SQL built-in functions.

LIKE and the elements formerly called "comparison operators" and "logical operators" are now documented as SQL conditions.

The chapters containing all SQL statements (formerly Chapters 7 through 10) have been divided into ten chapters for printing purposes.

Structural Changes in the SQL Reference in Oracle8*i*

The chapter containing all SQL statements (formerly Chapter 7) has been divided into four chapters for printing purposes.

Users familiar with the Release 8.0 documentation will find that the following sections have been moved or renamed:

- The section "Format Models" now appears in Chapter 2 on page 2-62.
- Chapter 3 has been divided into several smaller chapters:
  - Chapter 3, "Operators"
  - Chapter 6, "Functions"
  - Chapter 4, "Expressions". The last section, "Queries and Subqueries" on page 5-26, provides background for the syntactic and semantic information in SELECT on page 18-4.
- A new chapter, Chapter 8, "About SQL Statements", has been added to help you find the correct SQL statement for a particular task.
- The *archive_log_clause* is no longer a separate section, but has been incorporated into ALTER SYSTEM on page 10-22.
- The *deallocate_unused_clause* is no longer a separate section, but has been incorporated into ALTER TABLE on page 11-2, ALTER CLUSTER on page 9-7, and ALTER INDEX on page 9-64.

- The *disable_clause* is no longer a separate section, but has been incorporated into CREATE TABLE on page 15-7 and ALTER TABLE on page 11-2.
- The *drop_clause* is no longer a separate section. It has become the *drop_constraint_clause* of the ALTER TABLE statement (to distinguish it from the new *drop_column_clause* of that statement). See ALTER TABLE on page 11-2.
- The *enable_clause* is no longer a separate section, but has been incorporated into CREATE TABLE on page 15-7 and ALTER TABLE on page 11-2.
- The *parallel_clause* is no longer a separate section. The clause has been simplified, and has been incorporated into the various statements where it is relevant.
- The *recover_clause* is no longer a separate section. Recovery functionality has been enhanced, and because it is always implemented through the ALTER DATABASE statement, it has been incorporated into that section. See ALTER DATABASE on page 9-13.
- The sections on **snapshots** and **snapshot logs** have been moved and renamed. Snapshot functionality has been greatly enhanced, and these objects are now called **materialized views**. See CREATE MATERIALIZED VIEW on page 14-5, ALTER MATERIALIZED VIEW on page 9-92, DROP MATERIALIZED VIEW on page 16-83, CREATE MATERIALIZED VIEW LOG on page 14-32, ALTER MATERIALIZED VIEW LOG on page 9-112, and DROP MATERIALIZED VIEW LOG on page 16-85.
- The section on **subqueries** has now been combined with the SELECT statement. See SELECT on page 18-4.
- The two SQL statements GRANT *object_privileges* and GRANT *system_privileges_and_roles* have been combined into one GRANT statement. See GRANT on page 17-29.
- The two SQL statements REVOKE *schema_object_privileges* and REVOKE *system_privileges_and_roles* have been combined into one REVOKE statement. See REVOKE on page 17-89.
- The two SQL statements AUDIT *sql_statements* and AUDIT *schema_objects* have been combined into one AUDIT statement. See AUDIT on page 12-52.

- The two SQL statements `NOAUDIT` *`sql_statements`* and `NOAUDIT` *`schema_objects`* have been combined into one `NOAUDIT` statement. See NOAUDIT on page 17-82.

Related Documentation

For more information, see these Oracle resources:

- *PL/SQL User's Guide and Reference* for information on PL/SQL, Oracle's procedural language extension to SQL
- *Pro*C/C++ Precompiler Programmer's Guide*, *SQL*Module for Ada Programmer's Guide*, and the *Pro*COBOL Precompiler Programmer's Guide* for detailed descriptions of Oracle embedded SQL

Many of the examples in this book use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

`http://www.oraclebookshop.com/`

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://otn.oracle.com/admin/account/membership.html`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://otn.oracle.com/docs/index.htm`

To access the database documentation search engine directly, please visit

`http://tahiti.oracle.com`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width) font`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

Convention	Meaning	Example
`lowercase monospace (fixed-width) font`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase italic monospace (fixed-width) font`	Lowercase italic monospace font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
. . .	Horizontal ellipsis points indicate either: ■ That we have omitted parts of the code that are not directly related to the example ■ That you can repeat a portion of the code	`CREATE TABLE ... AS subquery;` `SELECT col1, col2, ... , coln FROM employees;`

.
.
.
	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	`SQL> SELECT NAME FROM V$DATAFILE;`
		`NAME`
		`----------`
		`/fsl/dbs/tbs_01.dbf`
		`/fs1/dbs/tbs_02.dbf`
		`.`
		`.`
		`.`
		`/fsl/dbs/tbs_09.dbf`
		`9 rows selected.`

Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
Italics	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/system_password` `DB_NAME = database_name`

UPPERCASE	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

Convention	Meaning	Example

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		
		`sqlplus hr/hr`
		
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle Corporation is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation This documentation may contain links to Web sites of other companies or organizations that Oracle Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

What's New in SQL Reference?

This section describes new features of Oracle9*i* release 2 and provides pointers to additional information. New features information from previous releases is also retained to help those users upgrading to the current release.

The following sections describe the new features in Oracle *SQL Reference*:

- Oracle9i Release 2 (9.2) New Features in the SQL Reference
- Oracle9i Release 1 (9.0.1) New Features in the SQL Reference
- Oracle8i New Features in SQL Reference

Oracle9i Release 2 (9.2) New Features in the SQL Reference

The following built-in conditions are new to this release:

- EQUALS_PATH on page 5-13
- UNDER_PATH on page 5-20

The following built-in expression is enhanced in this release:

- The syntax for type constructor expressions now allows creation of new user-defined constructors (see "Type Constructor Expressions" on page 4-13).

The following built-in functions are new to this release:

- DEPTH on page 6-55
- EXTRACTVALUE on page 6-63
- PATH on page 6-112
- UPDATEXML on page 6-194
- XMLAGG on page 6-207
- XMLCONCAT on page 6-209
- XMLCOLATTVAL on page 6-208
- XMLELEMENT on page 6-211
- XMLFOREST on page 6-214
- XMLSEQUENCE on page 6-215
- XMLTRANSFORM on page 6-216

The following privileges are new or enhanced in this release:

- `DEBUG CONNECT SESSION` system privilege on page 17-37 (new)
- `DEBUG ANY PROCEDURE` system privilege on page 17-37 (new)
- `DEBUG` object privilege on page 17-47 (new)
- `EXECUTE` object privilege on procedures, packages, libraries, and object types (enhanced)
- `GRANT ANY OBJECT PRIVILEGE` system privilege on page 17-44

The following top-level SQL statements are new or enhanced in this release:

- ALTER DATABASE on page 9-13 has new syntax for managing standby databases, for managed standby recovery, and for logical standby.
- ALTER OPERATOR on page 9-119 lets you recompile an existing user-defined operator.
- ALTER TABLE on page 11-2 contains new clauses that let you rename a column or a constraint.
- CREATE DATABASE on page 13-22:
  - Contains two new clauses for assigning passwords to the SYS and SYSTEM users
  - Lets you create a locally managed SYSTEM tablespace
- CREATE SYNONYM on page 15-2 now allows creation of synonyms for object types.
- CREATE TABLE on page 15-7:
  - Allows creation of a default list partition to capture rows that do not fall within any of the other list partitions
  - Allows creation of range-list composite-partitioned tables
  - Contains syntax for creating a table of type XMLType and for creating range-list composite-partitioned tables
  - Allows data compression of data in table and partition segments
- CREATE TYPE on page 16-3 allows creation of object types with of NCHAR, NVARCHAR2, and NCLOB attributes.
- CREATE VIEW on page 16-39 now contains syntax for creating an XML view by transforming a table of type XMLType.
- SELECT on page 18-4 provides syntax for "flashback queries", which let you query data at a specified system change number or time in the past.

Oracle9*i* Release 1 (9.0.1) New Features in the SQL Reference

The following built-in datatypes were new or modified in this release:

- Oracle provides SQL-based interfaces for defining new types when the built-in or ANSI-supported types are not sufficient. See "Oracle-Supplied Types" on page 2-40.

- "CHAR Datatype" on page 2-9 can take the CHAR or BYTE parameter to indicate character or byte semantics, respectively.
- "INTERVAL YEAR TO MONTH Datatype" on page 2-23 and "INTERVAL DAY TO SECOND Datatype" on page 2-24 are provided for additional datetime functionality
- "TIMESTAMP Datatype" on page 2-21 is provided for additional datetime functionality.
- "VARCHAR2 Datatype" on page 2-11 can take the CHAR or BYTE parameter to indicate character or byte semantics, respectively.

The following expression forms were new or enhanced in this release:

- "CASE Expressions" on page 4-6 (enhanced with searched case expression)
- "CURSOR Expressions" on page 4-7 (enhanced; they can be passed as REF CURSOR arguments to functions)
- "Datetime Expressions" on page 4-9 (new)
- "INTERVAL Expressions" on page 4-11 (new)
- "Scalar Subquery Expressions" on page 4-13 (new)

The following condition was new in this release:

- IS OF type Conditions on page 5-19

The following built-in functions were new to this release:

- ASCIISTR on page 6-18
- BIN_TO_NUM on page 6-23
- COALESCE on page 6-31
- COMPOSE on page 6-32
- CURRENT_DATE on page 6-47
- CURRENT_TIMESTAMP on page 6-48
- DBTIMEZONE on page 6-49
- DECOMPOSE on page 6-51
- EXISTSNODE on page 6-59
- EXTRACT (datetime) on page 6-61
- EXTRACT (XML) on page 6-62

- TREAT on page 6-188
- TZ_OFFSET on page 6-192
- UNISTR on page 6-194
- WIDTH_BUCKET on page 6-205

The following functions were enhanced for this release:

- INSTR on page 6-75
- LENGTH on page 6-85
- SUBSTR on page 6-149

The following privileges were new to this release:

- EXEMPT ACCESS POLICY system privilege on page 17-44
- RESUMABLE system privilege on page 17-44
- SELECT ANY DICTIONARY system privilege on page 17-45
- UNDER ANY TYPE system privilege on page 17-43
- UNDER ANY VIEW system privilege on page 17-43
- UNDER object privilege on page 17-47

The following top-level SQL statements were new to this release:

- CREATE PFILE on page 14-60
- CREATE SPFILE on page 14-92
- MERGE on page 17-78

The following SQL statements had new syntax:

- ALTER DATABASE on page 9-13 has new syntax that lets you end a "hot backup" procedure while the database is mounted. It also has new syntax related to standby databases.
- ALTER INDEX on page 9-64 lets you gather statistics on index usage.
- ALTER OUTLINE on page 9-120 allows modification of both public and private outlines.
- ALTER ROLE on page 9-136 lets you identify a role using an application-specified package.

- ALTER SESSION on page 10-2 lets you specify whether statements issued during the session can be suspended under some conditions.
- ALTER SYSTEM on page 10-22 has extended SET clause and new RESET clause; lets you put the database in quiesed state.
- ALTER TABLE on page 11-2 allows partitioning by a list of specified values.
- ALTER TYPE on page 12-6 lets you modify the attribute or method definition of an object type.
- ALTER VIEW on page 12-30 lets you add constraints to views.
- ANALYZE on page 12-33 now has ONLINE and OFFLINE clauses as part of the VALIDATE STRUCTURE syntax. In addition, you can now choose whether to collect standard statistics, user-defined statistics, or both.
- *constraints* on page 7-5 has been enhanced to facilitate index handling when dropping or disabling constraints.
- CREATE CONTEXT on page 13-12 has added syntax to let you initialize the context from the LDAP directory or from an OCI interface and to make the context accessible throughout an instance.
- CREATE CONTROLFILE on page 13-15 allows creation of Oracle-managed files.
- CREATE DATABASE on page 13-22 lets you create default temporary tablespaces when you create the database; lets you create undo tablespaces.
- CREATE FUNCTION on page 13-49 lets you create pipelined and parallel table functions and user-defined aggregate functions.
- CREATE OUTLINE on page 14-46 allows creation of both public and private outlines.
- CREATE ROLE on page 14-77 lets you identify a role using an application-specified package.
- CREATE TABLE on page 15-7 allows creation of external tables (tables whose data is outside the database); allows creation of Oracle-managed files; allows partitioning by a list of specified values.
- CREATE TABLESPACE on page 15-80 allows for segment space management by bitmaps as well as by free lists; allows creation of Oracle-managed files; lets you create undo tablespaces.
- CREATE TEMPORARY TABLESPACE on page 15-92 allows creation of Oracle-managed files.

- CREATE TYPE on page 16-3 lets you create subtypes.
- CREATE VIEW on page 16-39 lets you create subviews of object views; lets you define constraints on views.
- DROP TABLESPACE on page 17-10 has added syntax that lets you drop operating system files when you drop the contents from a dropped tablespace.
- *file_specification* on page 7-39 allows creation of Oracle-managed files.
- INSERT on page 17-54 has added syntax that lets you insert default column values.
- SELECT on page 18-4 lets you specify multiple groupings in the GROUP BY clause, for selective analysis across multiple dimensions; lets you assign names to subquery blocks; has new ANSI-compliant join syntax.
- SET TRANSACTION on page 18-50 lets you specify a name for a transaction.
- UPDATE on page 18-59 has added syntax that lets you update to default column values.

Oracle8*i* New Features in SQL Reference

The following SQL functions were new to this version:

- BITAND on page 6-24
- CORR on page 6-35
- COVAR_POP on page 6-40
- COVAR_SAMP on page 6-42
- CUME_DIST on page 6-45
- DENSE_RANK on page 6-53
- FIRST_VALUE on page 6-66
- LAG on page 6-77
- LAST_VALUE on page 6-81
- LEAD on page 6-83
- NTILE on page 6-106
- NUMTOYMINTERVAL on page 6-109
- NUMTODSINTERVAL on page 6-108

- NVL2 on page 6-111
- PERCENT_RANK on page 6-113
- RANK on page 6-120
- RATIO_TO_REPORT on page 6-122
- REGR_ (Linear Regression) Functions on page 6-126
- STDDEV_POP on page 6-146
- STDDEV_SAMP on page 6-148
- VAR_POP on page 6-199
- VAR_SAMP on page 6-201

The following top-level SQL statements were new to Release 8.1.5:

- ALTER DIMENSION on page 9-58
- ALTER JAVA on page 9-89
- ALTER OUTLINE on page 9-120
- ASSOCIATE STATISTICS on page 12-48
- CALL on page 12-66
- CREATE CONTEXT on page 13-12
- CREATE DIMENSION on page 13-41
- CREATE INDEXTYPE on page 13-91
- CREATE JAVA on page 13-94
- CREATE OPERATOR on page 14-42
- CREATE OUTLINE on page 14-46
- CREATE TEMPORARY TABLESPACE on page 15-92
- DISASSOCIATE STATISTICS on page 16-64
- DROP CONTEXT on page 16-69
- DROP DIMENSION on page 16-71
- DROP INDEXTYPE on page 16-78
- DROP JAVA on page 16-80
- DROP OPERATOR on page 16-87

- DROP OUTLINE on page 16-89

In addition, the following features were enhanced:

- The aggregate functions have expanded functionality. See "Aggregate Functions" on page 6-7.
- When specifying LOB storage parameters, you can now specify caching of LOBs for read-only purposes. See CREATE TABLE on page 15-7.
- The section on Expressions now contains a new expression. See "CASE Expressions" on page 4-6.
- Subqueries can now be unnested. See "Unnesting of Nested Subqueries" on page 8-14.

1

Introduction

Structured Query Language (SQL) is the set of statements with which all programs and users access data in an Oracle database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications in turn must use SQL when executing the user's request. This chapter provides background information on SQL as used by most database systems.

This chapter contains these topics:

History of SQL

Dr. E. F. Codd published the paper, "A Relational Model of Data for Large Shared Data Banks", in June 1970 in the Association of Computer Machinery (ACM) journal, *Communications of the ACM*. Codd's model is now accepted as the definitive model for relational database management systems (RDBMS). The language, Structured English Query Language ("SEQUEL") was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel"). In 1979, Relational Software, Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

SQL Standards

Oracle Corporation strives to comply with industry-accepted standards and participates actively in SQL standards committees. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Standards Organization (ISO), which is affiliated with the International Electrotechnical Commission (IEC). Both ANSI and the ISO / IEC have accepted SQL as the standard language for relational databases. When a new SQL standard is simultaneously published by these organizations, the names of the standards conform to conventions used by the organization, but the standards are technically identical.

The latest SQL standard was adopted in July 1999 and is often called SQL:99. The formal names of this standard are:

- ANSI X3.135-1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")
- ISO / IEC 9075:1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")

> **See Also:** Appendix B, "Oracle and Standard SQL" for a detailed description of Oracle's conformance to the SQL:99 standards

How SQL Works

The strengths of SQL provide benefits for all types of users, including application programmers, database administrators, managers, and end users. Technically speaking, SQL is a data sublanguage. The purpose of SQL is to provide an interface to a relational database such as Oracle, and all SQL statements are instructions to the database. In this SQL differs from general-purpose programming languages like C and BASIC. Among the features of SQL are the following:

- It processes sets of data as groups rather than as individual units.
- It provides automatic navigation to the data.
- It uses statements that are complex and powerful individually, and that therefore stand alone. Flow-control statements were not part of SQL originally, but they are found in the recently accepted optional part of SQL, ISO / IEC 9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL / SQL extension to SQL is similar to PSM.

Essentially, SQL lets you work with data at the logical level. You need to be concerned with the implementation details only when you want to manipulate the data. For example, to retrieve a set of rows from a table, you define a condition used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. You need not deal with the rows one by one, nor do you have to worry about how they are physically stored or retrieved. All SQL statements use the **optimizer**, a part of Oracle that determines the most efficient means of accessing the specified data. Oracle also provides techniques that you can use to make the optimizer perform its job better.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the preceding tasks in one consistent language.

Common Language for All Relational Databases

All major relational database management systems support SQL, so you can transfer all skills you have gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

Embedded SQL

Embedded SQL refers to the use of standard SQL statements embedded within a procedural programming language. The embedded SQL statements are documented in the Oracle precompiler books.

Embedded SQL is a collection of these statements:

- All SQL commands, such as SELECT and INSERT, available with SQL with interactive tools
- Dynamic SQL execution commands, such as PREPARE and OPEN, which integrate the standard SQL statements with a procedural programming language

Embedded SQL also includes extensions to some standard SQL statements. Embedded SQL is supported by the Oracle precompilers. The Oracle precompilers interpret embedded SQL statements and translate them into statements that can be understood by procedural language compilers.

Each of these Oracle precompilers translates embedded SQL programs into a different procedural language:

- Pro*C/C++ precompiler
- Pro*COBOL precompiler

> **See Also:** *Pro*C/C++ Precompiler Programmer's Guide* and *Pro*COBOL Precompiler Programmer's Guide* for a definition of the Oracle precompilers and embedded SQL statements

Lexical Conventions

The following lexical conventions for issuing SQL statements apply specifically to Oracle's implementation of SQL, but are generally acceptable in other SQL implementations.

When you issue a SQL statement, you can include one or more tabs, carriage returns, spaces, or comments anywhere a space occurs within the definition of the statement. Thus, Oracle evaluates the following two statements in the same manner:

```
SELECT last_name,salary*12,MONTHS_BETWEEN(hire_date, SYSDATE)
   FROM employees;

SELECT last_name,
     salary * 12,
          MONTHS_BETWEEN( hire_date, SYSDATE )
```

```
FROM employees;
```

Case is insignificant in reserved words, keywords, identifiers and parameters. However, case is significant in text literals and quoted names.

See Also: "Text Literals" on page 2-54 for a syntax description

Tools Support

Most (but not all) Oracle tools support all features of Oracle SQL. This reference describes the complete functionality of SQL. If the Oracle tool that you are using does not support this complete functionality, you can find a discussion of the restrictions in the manual describing the tool, such as *SQL*Plus User's Guide and Reference*.

2

Basic Elements of Oracle SQL

This chapter contains reference information on the basic elements of Oracle SQL. These elements are the simplest building blocks of SQL statements. Therefore, before using the statements described in Chapter 9 through Chapter 18, you should familiarize yourself with the concepts covered in this chapter, as well as in Chapter 3, "Operators", Chapter 4, "Expressions", Chapter 6, "Functions", and Chapter 8, "SQL Queries and Subqueries".

This chapter contains these sections:

- Datatypes
- Literals
- Format Models
- Nulls
- Pseudocolumns
- Comments
- Database Objects
- Schema Object Names and Qualifiers
- Syntax for Schema Objects and Parts in SQL Statements

Datatypes

Each value manipulated by Oracle has a **datatype**. A value's datatype associates a fixed set of properties with the value. These properties cause Oracle to treat values of one datatype differently from values of another. For example, you can add values of NUMBER datatype, but not values of RAW datatype.

When you create a table or cluster, you must specify a datatype for each of its columns. When you create a procedure or stored function, you must specify a datatype for each of its arguments. These datatypes define the domain of values that each column can contain or each argument can have. For example, DATE columns cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'. Each value subsequently placed in a column assumes the column's datatype. For example, if you insert '01-JAN-98' into a DATE column, then Oracle treats the '01-JAN-98' character string as a DATE value after verifying that it translates to a valid date.

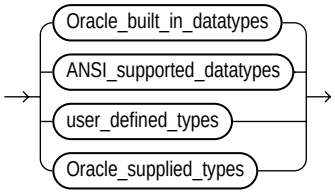
Oracle provides a number of built-in datatypes as well as several categories for user-defined types that can be used as datatypes. The syntax of Oracle datatypes appears in the diagrams that follow. The text of this section is divided into the following sections:

- Oracle Built-in Datatypes
- ANSI, DB2, and SQL/DS Datatypes
- User-Defined Types
- Oracle-Supplied Types

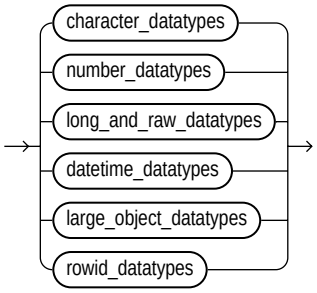
Note: The Oracle precompilers recognize other datatypes in embedded SQL programs. These datatypes are called **external datatypes** and are associated with host variables. Do not confuse built-in datatypes and user-defined types with external datatypes. For information on external datatypes, including how Oracle converts between them and built-in datatypes or user-defined types, see *Pro*COBOL Precompiler Programmer's Guide*, and *Pro*C/C++ Precompiler Programmer's Guide*.

---

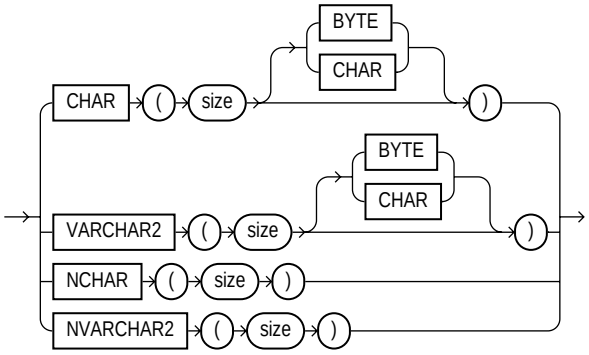
datatypes::=



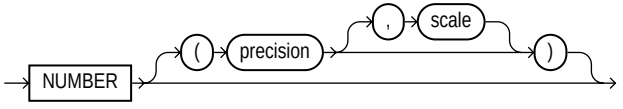
Oracle_built_in_datatypes::=



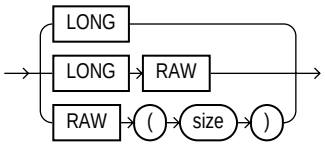
character_datatypes::=



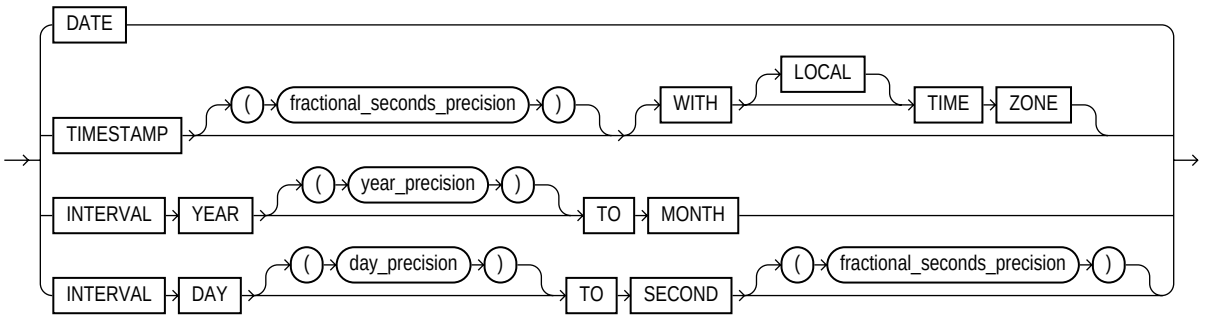
number_datatypes::=



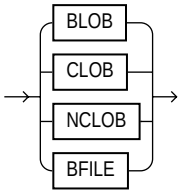
long_and_raw_datatypes::=



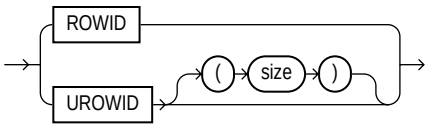
datetime_datatypes::=



large_object_datatypes::=

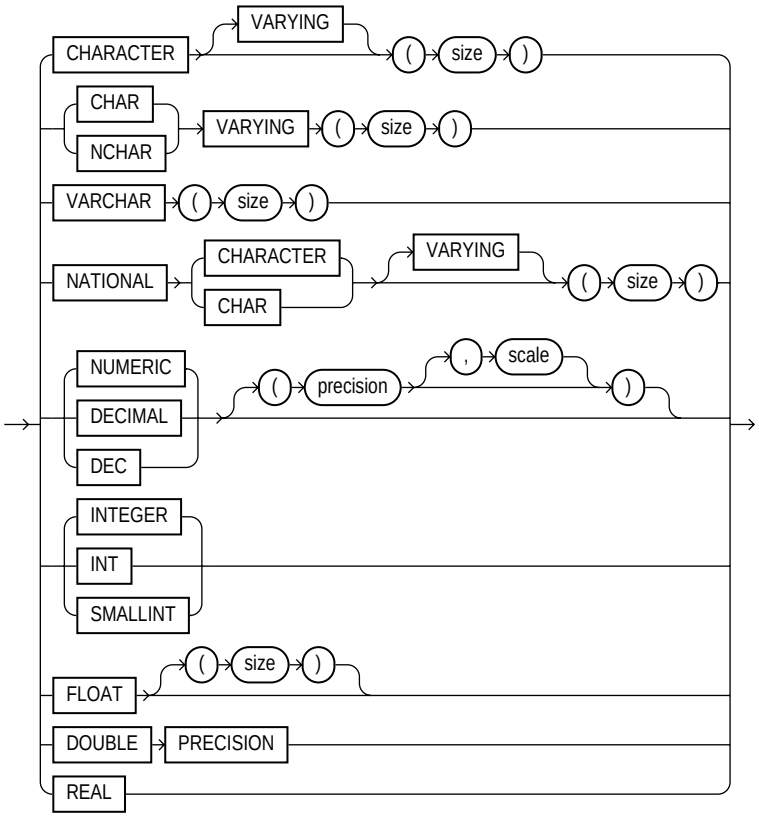


rowid_datatypes::=

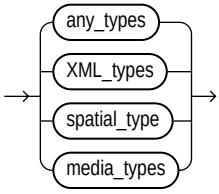


The ANSI-supported datatypes appear in the figure that follows. Table 2–6 on page 2-36 shows the mapping of ANSI-supported datatypes to Oracle built-in datatypes.

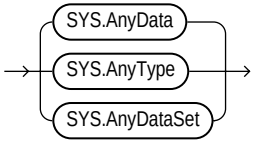
ANSI_supported_datatypes::=



Oracle_supplied_types::=

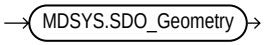


any_types::=



XML_types::=

spatial_type::=



media_types::=

Oracle Built-in Datatypes

Table 2–1 summarizes Oracle built-in datatypes.

Table 2–1 Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
1	VARCHAR2(*size*) [BYTE | CHAR]	Variable-length character string having maximum length *size* bytes or characters. Maximum *size* is 4000 bytes, and minimum is 1 byte or 1 character. You must specify *size* for VARCHAR2. BYTE indicates that the column will have byte length semantics; CHAR indicates that the column will have character semantics.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

Table 2–1 (Cont.) Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
1	NVARCHAR2(*size*)	Variable-length character string having maximum length *size* characters. Maximum *size* is determined by the national character set definition, with an upper limit of 4000 bytes. You must specify *size* for NVARCHAR2.
2	NUMBER(*p*,*s*)	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127.
8	LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes.
12	DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD.
180	TIMESTAMP (*fractional_seconds_precision*)	Year, month, and day values of date, as well as hour, minute, and second values of time, where *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values of *fractional_seconds_precision* are 0 to 9. The default is 6.
181	TIMESTAMP (*fractional_seconds_precision*) WITH TIME ZONE	All values of TIMESTAMP as well as time zone displacement value, where *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.
231	TIMESTAMP (*fractional_seconds_precision*) WITH LOCAL TIME ZONE	All values of TIMESTAMP WITH TIME ZONE, with the following exceptions: ■ Data is normalized to the database time zone when it is stored in the database. ■ When the data is retrieved, users see the data in the session time zone.
182	INTERVAL YEAR (*year_precision*) TO MONTH	Stores a period of time in years and months, where *year_precision* is the number of digits in the YEAR datetime field. Accepted values are 0 to 9. The default is 2.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

Table 2–1 (Cont.) Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
183	INTERVAL DAY (*day_precision*) TO SECOND (*fractional_seconds_precision*)	Stores a period of time in days, hours, minutes, and seconds, where ■ *day_precision* is the maximum number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2. ■ *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND field. Accepted values are 0 to 9. The default is 6.
23	RAW(*size*)	Raw binary data of length *size* bytes. Maximum *size* is 2000 bytes. You must specify *size* for a RAW value.
24	LONG RAW	Raw binary data of variable length up to 2 gigabytes.
69	ROWID	Base 64 string representing the unique address of a row in its table. This datatype is primarily for values returned by the ROWID pseudocolumn.
208	UROWID [(*size*)]	Base 64 string representing the logical address of a row of an index-organized table. The optional *size* is the size of a column of type UROWID. The maximum size and default is 4000 bytes.
96	CHAR(*size*)[BYTE | CHAR]	Fixed-length character data of length *size* bytes. Maximum *size* is 2000 bytes. Default and minimum *size* is 1 byte. BYTE and CHAR have the same semantics as for VARCHAR2.
96	NCHAR(*size*)	Fixed-length character data of length *size* characters. Maximum *size* is determined by the national character set definition, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character.
112	CLOB	A character large object containing single-byte characters. Both fixed-width and variable-width character sets are supported, both using the CHAR database character set. Maximum size is 4 gigabytes.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

Table 2–1 (Cont.) Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
112	NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the NCHAR database character set. Maximum size is 4 gigabytes. Stores national character set data.
113	BLOB	A binary large object. Maximum size is 4 gigabytes.
114	BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned by the DUMP function.

Character Datatypes

Character datatypes store character (alphanumeric) data, which are words and free-form text, in the database character set or national character set. They are less restrictive than other datatypes and consequently have fewer properties. For example, character columns can store all alphanumeric values, but NUMBER columns can store only numeric values.

Character data is stored in strings with byte values corresponding to one of the character sets, such as 7-bit ASCII or EBCDIC, specified when the database was created. Oracle supports both single-byte and multibyte character sets.

These datatypes are used for character data:

- CHAR Datatype
- NCHAR Datatype
- NVARCHAR2 Datatype
- VARCHAR2 Datatype

CHAR Datatype

The CHAR datatype specifies a fixed-length character string. Oracle subsequently ensures that all values stored in that column have the length specified by *size*. If you insert a value that is shorter than the column length, then Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, then Oracle returns an error.

The default length for a CHAR column is 1 byte and the maximum allowed is 2000 bytes. A 1-byte string can be inserted into a CHAR(10) column, but the string is blank-padded to 10 bytes before it is stored.

When you create a table with a CHAR column, by default you supply the column length in bytes. The BYTE qualifier is the same as the default. If you use the CHAR qualifier, for example CHAR(10 CHAR), then you supply the column length in characters. A character is technically a codepoint of the database character set. Its size can range from 1 byte to 4 bytes, depending on the database character set. The BYTE and CHAR qualifiers override the semantics specified by the NLS_LENGTH_SEMANTICS parameter, which has a default of byte semantics.

Note: To ensure proper data conversion between databases with different character sets, you must ensure that CHAR data consists of well-formed strings. See *Oracle9i Database Globalization Support Guide* for more information on character set support.

See Also: "Datatype Comparison Rules" on page 2-45 for information on comparison semantics

NCHAR Datatype

Beginning with Oracle9*i*, the NCHAR datatype is redefined to be a Unicode-only datatype. When you create a table with an NCHAR column, you define the column length in characters. You define the national character set when you create your database.

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NCHAR refer to the number of characters. The maximum column size allowed is 2000 bytes.

If you insert a value that is shorter than the column length, then Oracle blank-pads the value to column length. You cannot insert a CHAR value into an NCHAR column, nor can you insert an NCHAR value into a CHAR column.

The following example compares the col1 column of tab1 with national character set string 'NCHAR literal':

```
SELECT translated_description from product_descriptions
   WHERE translated_name = N'LCD Monitor 11/PM';
```

See Also: *Oracle9i Database Globalization Support Guide* for information on Unicode datatype support

NVARCHAR2 Datatype

Beginning with Oracle9*i*, the NVARCHAR2 datatype is redefined to be a Unicode-only datatype. When you create a table with an NVARCHAR2 column, you supply the maximum number of characters it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length.

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NVARCHAR2 refer to the number of characters. The maximum column size allowed is 4000 bytes.

> **See Also:** *Oracle9i Database Globalization Support Guide* for information on Unicode datatype support

VARCHAR2 Datatype

The VARCHAR2 datatype specifies a variable-length character string. When you create a VARCHAR2 column, you supply the maximum number of bytes or characters of data that it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length. If you try to insert a value that exceeds the specified length, then Oracle returns an error.

You must specify a maximum length for a VARCHAR2 column. This maximum must be at least 1 byte, although the actual string stored is permitted to be a zero-length string (''). You can use the CHAR qualifier, for example VARCHAR2(10 CHAR), to give the maximum length in characters instead of bytes. A character is technically a codepoint of the database character set. CHAR and BYTE qualifiers override the setting of the NLS_LENGTH_SEMANTICS parameter, which has a default of bytes. The maximum length of VARCHAR2 data is 4000 bytes. Oracle compares VARCHAR2 values using nonpadded comparison semantics.

Note: To ensure proper data conversion between databases with different character sets, you must ensure that VARCHAR2 data consists of well-formed strings. See *Oracle9i Database Globalization Support Guide* for more information on character set support.

> **See Also:** "Datatype Comparison Rules" on page 2-45 for information on comparison semantics

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. Oracle recommends that you use VARCHAR2 rather than VARCHAR. In the future, VARCHAR might be defined as a separate datatype used for variable-length character strings compared with different comparison semantics.

NUMBER Datatype

The NUMBER datatype stores zero, positive, and negative fixed and floating-point numbers with magnitudes between 1.0×10^{-130} and $9.9...9 \times 10^{125}$ (38 nines followed by 88 zeroes) with 38 digits of precision. If you specify an arithmetic expression whose value has a magnitude greater than or equal to 1.0×10^{126}, then Oracle returns an error.

Specify a fixed-point number using the following form:

```
NUMBER(p,s)
```

where:

- *p* is the **precision**, or the total number of digits. Oracle guarantees the portability of numbers with precision ranging from 1 to 38.
- *s* is the **scale**, or the number of digits to the right of the decimal point. The scale can range from -84 to 127.

Specify an integer using the following form:

```
NUMBER(p)
```

This represents a fixed-point number with precision p and scale 0 and is equivalent to `NUMBER(p,0)`.

Specify a floating-point number using the following form:

```
NUMBER
```

The absence of precision and scale designators specifies the maximum range and precision for an Oracle number.

See Also: "Floating-Point Numbers" on page 2-14

Scale and Precision

Specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed

length. If a value exceeds the precision, then Oracle returns an error. If a value exceeds the scale, then Oracle rounds it.

Table 2–2 show how Oracle stores data using different precisions and scales.

Table 2–2 Storage of Scale and Precision

Actual Data	Specified As	Stored As
7456123.89	NUMBER	7456123.89
7456123.89	NUMBER(9)	7456124
7456123.89	NUMBER(9,2)	7456123.89
7456123.89	NUMBER(9,1)	7456123.9
7456123.89	NUMBER(6)	exceeds precision
7456123.89	NUMBER(7,-2)	7456100
7456123.89	NUMBER(7,2)	exceeds precision

Negative Scale

If the scale is negative, then the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

Scale Greater than Precision

You can specify a scale that is greater than precision, although it is uncommon. In this case, the precision specifies the maximum number of digits to the right of the decimal point. As with all number datatypes, if the value exceeds the precision, then Oracle returns an error message. If the value exceeds the scale, then Oracle rounds the value. For example, a column defined as NUMBER(4,5) requires a zero for the first digit after the decimal point and rounds all values past the fifth digit after the decimal point. Table 2–3 show the effects of a scale greater than precision:

Table 2–3 Scale Greater Than Precision

Actual Data	Specified As	Stored As
.01234	NUMBER(4,5)	.01234
.00012	NUMBER(4,5)	.00012
.000127	NUMBER(4,5)	.00013
.0000012	NUMBER(2,7)	.0000012

Table 2–3 Scale Greater Than Precision

Actual Data	Specified As	Stored As
.00000123	NUMBER(2,7)	.0000012

Floating-Point Numbers

Oracle lets you specify floating-point numbers, which can have a decimal point anywhere from the first to the last digit or can have no decimal point at all. An exponent may optionally be used following the number to increase the range (for example, 1.777 e^{-20}). A scale value is not applicable to floating-point numbers, because the number of digits that can appear after the decimal point is not restricted.

You can specify floating-point numbers with the range of values discussed in "NUMBER Datatype" on page 2-12. The format is defined in "Number Literals" on page 2-56. Oracle also supports the ANSI datatype FLOAT. You can specify this datatype using one of these syntactic forms:

- FLOAT specifies a floating-point number with decimal precision 38 or binary precision 126.
- FLOAT(b) specifies a floating-point number with binary precision *b*. The precision *b* can range from 1 to 126. To convert from binary to decimal precision, multiply *b* by 0.30103. To convert from decimal to binary precision, multiply the decimal precision by 3.32193. The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

LONG Datatype

LONG columns store variable-length character strings containing up to 2 gigabytes, or 2^{31}-1 bytes. LONG columns have many of the characteristics of VARCHAR2 columns. You can use LONG columns to store long text strings. The length of LONG values may be limited by the memory available on your computer.

Note: Oracle Corporation strongly recommends that you convert LONG columns to LOB columns as soon as possible. Creation of new LONG columns is scheduled for desupport.

LOB columns are subject to far fewer restrictions than LONG columns. Further, LOB functionality is enhanced in every release, whereas LONG functionality has been static for several releases. See the *modify_column_options* clause of ALTER TABLE on page 11-2 and TO_LOB on page 6-175 for more information on converting LONG columns to LOB.

You can reference LONG columns in SQL statements in these places:

- SELECT lists
- SET clauses of UPDATE statements
- VALUES clauses of INSERT statements

The use of LONG values is subject to some restrictions:

- A table can contain only one LONG column.
- You cannot create an object type with a LONG attribute.
- LONG columns cannot appear in WHERE clauses or in integrity constraints (except that they can appear in NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- A stored function cannot return a LONG value.
- You can declare a variable or argument of a PL/SQL program unit using the LONG datatype. However, you cannot then call the program unit from SQL.
- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.
- LONG and LONG RAW columns cannot be used in distributed SQL statements and cannot be replicated.
- If a table has both LONG and LOB columns, you cannot bind more than 4000 bytes of data to both the LONG and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the LONG or the LOB column.

- A table with LONG columns cannot be stored in a tablespace with automatic segment-space management.

LONG columns cannot appear in certain parts of SQL statements:

- GROUP BY clauses, ORDER BY clauses, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- The UNIQUE operator of a SELECT statement
- The column list of a CREATE CLUSTER statement
- The CLUSTER clause of a CREATE MATERIALIZED VIEW statement
- SQL built-in functions, expressions, or conditions
- SELECT lists of queries containing GROUP BY clauses
- SELECT lists of subqueries or queries combined by the UNION, INTERSECT, or MINUS set operators
- SELECT lists of CREATE TABLE ... AS SELECT statements
- ALTER TABLE ... MOVE statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG datatype in the following manner:

- A SQL statement within a trigger can insert data into a LONG column.
- If data from a LONG column can be converted to a constrained datatype (such as CHAR and VARCHAR2), a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG datatype.
- :NEW and :OLD cannot be used with LONG columns.

You can use the Oracle Call Interface functions to retrieve a portion of a LONG value from the database.

See Also: *Oracle Call Interface Programmer's Guide*

Datetime and Interval Datatypes

The datetime datatypes are DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE. Values of datetime datatypes are sometimes called "datetimes". The interval datatypes are INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND. Values of interval datatypes are sometimes called "intervals".

Both datetimes and intervals are made up of fields. The values of these fields determine the value of the datatype. Table 2–4 lists the datetime fields and their possible values for datetimes and intervals.

Table 2–4 Datetime Fields and Values

Datetime Field	Valid Values for Datetime	Valid Values for INTERVAL
YEAR	-4712 to 9999 (excluding year 0)	Any positive or negative integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the current NLS calendar parameter)	Any positive or negative integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where "9(n)" is the precision of time fractional seconds The "9(n)" portion is not applicable for DATE.	0 to 59.9(n), where “9(n)” is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (This range accommodates daylight savings time changes.) Not applicable for DATE.	Not applicable
TIMEZONE_MINUTE	00 to 59. Not applicable for DATE.	Not applicable
TIMEZONE_REGION	Query the TZNAME column of the V$TIMEZONE_NAMES data dictionary view. Not applicable for DATE.	Not applicable
TIMEZONE_ABBR	Query the TZABBREV column of the V$TIMEZONE_NAMES data dictionary view. Not applicable for DATE.	Not applicable

Note: To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the time zones have not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.

DATE Datatype

The DATE datatype stores date and time information. Although date and time information can be represented in both character and number datatypes, the DATE datatype has special associated properties. For each DATE value, Oracle stores the following information: century, year, month, date, hour, minute, and second.

You can specify a date value as a literal, or you can convert a character or numeric value to a date value with the TO_DATE function. To specify a date as a literal, you must use the Gregorian calendar. You can specify an ANSI date literal, as shown in this example:

```
DATE '1998-12-25'
```

The ANSI date literal contains no time portion, and must be specified in exactly this format ('YYYY-MM-DD'). Alternatively you can specify an Oracle date literal, as in the following example:

```
TO_DATE('98-DEC-25:17:30','YY-MON-DD:HH24:MI')
```

The default date format for an Oracle date literal is specified by the initialization parameter NLS_DATE_FORMAT. This example date format includes a two-digit number for the day of the month, an abbreviation of the month name, the last two digits of the year, and a 24-hour time designation.

Oracle automatically converts character values that are in the default date format into date values when they are used in date expressions.

If you specify a date value without a time component, then the default time is 12:00:00 AM (midnight). If you specify a date value without a date, then the default date is the first day of the current month.

Oracle DATE columns always contain both the date and time fields. If your queries use a date format without a time portion, then you must ensure that the time fields in the DATE column are set to zero (that is, midnight). Otherwise, Oracle may not

return the query results you expect. Here are some examples that assume a table my_table with a number column row_num and a DATE column datecol:

```
INSERT INTO my_table VALUES (1, SYSDATE);
INSERT INTO my_table VALUES (2, TRUNC(SYSDATE));

SELECT * FROM my_table;

   ROW_NUM DATECOL
---------- ---------
         1 04-OCT-00
         2 04-OCT-00

SELECT * FROM my_table
   WHERE datecol = TO_DATE('04-OCT-00','DD-MON-YY');

   ROW_NUM DATECOL
---------- ---------
         2 04-OCT-00
```

If you know that the time fields of your DATE column are set to zero, then you can query your DATE column as shown in the immediately preceding example, or by using the DATE literal:

```
SELECT * FROM my_table WHERE datecol = DATE '2000-10-04';
```

However, if the DATE column contains nonzero time fields, then you must filter out the time fields in the query to get the correct result. For example:

```
SELECT * FROM my_table WHERE TRUNC(datecol) = DATE '2000-10-04';
```

Oracle applies the TRUNC function to each row in the query, so performance is better if you ensure the zero value of the time fields in your data. To ensure that the time fields are set to zero, use one of the following methods during inserts and updates:

- Use the TO_DATE function to mask out the time fields:

  ```
  INSERT INTO my_table VALUES
     (3, TO_DATE('4-APR-2000','DD-MON-YYYY'));
  ```

- Use the DATE literal:

  ```
  INSERT INTO my_table VALUES (4, '04-OCT-00');
  ```

- Use the TRUNC function:

  ```
  INSERT INTO my_table VALUES (5, TRUNC(SYSDATE));
  ```

The date function SYSDATE returns the current system date and time. The function CURRENT_DATE returns the current session date. For information on SYSDATE, the TO_* datetime functions, and the default date format, see Chapter 6, "Functions".

Date Arithmetic You can add and subtract number constants as well as other dates from dates. Oracle interprets number constants in arithmetic date expressions as numbers of days. For example, SYSDATE + 1 is tomorrow. SYSDATE - 7 is one week ago. SYSDATE + (10/1440) is ten minutes from now. Subtracting the hiredate column of the sample table employees from SYSDATE returns the number of days since each employee was hired. You cannot multiply or divide DATE values.

Oracle provides functions for many common date operations. For example, the ADD_MONTHS function lets you add or subtract months from a date. The MONTHS_BETWEEN function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31-day month.

Because each date contains a time component, most results of date operations include a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours.

See Also:

- "Datetime Functions" on page 6-5 for more information on date functions
- "Datetime/Interval Arithmetic" on page 2-24 for information on arithmetic involving other datetime and interval datatypes

---

Using Julian Dates A Julian date is the number of days since January 1, 4712 BC. Julian dates allow continuous dating from a common reference. You can use the date format model "J" with date functions TO_DATE and TO_CHAR to convert between Oracle DATE values and their Julian equivalents.

Example This statement returns the Julian equivalent of January 1, 1997:

```
SELECT TO_CHAR(TO_DATE('01-01-1997', 'MM-DD-YYYY'),'J')
    FROM DUAL;

TO_CHAR
--------
2450450
```

> **See Also:** "Selecting from the DUAL Table" on page 8-15 for a description of the DUAL table

TIMESTAMP Datatype

The TIMESTAMP datatype is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, plus hour, minute, and second values. Specify the TIMESTAMP datatype as follows:

```
TIMESTAMP [ (fractional_seconds_precision)]
```

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6. For example, you specify TIMESTAMP as a literal as follows:

```
TIMESTAMP'1997-01-31 09:26:50.124'
```

> **See Also:** TO_TIMESTAMP on page 6-182 for information on converting character data to TIMESTAMP data

TIMESTAMP WITH TIME ZONE Datatype

TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a **time zone displacement** in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). Specify the TIMESTAMP WITH TIME ZONE datatype as follows:

```
TIMESTAMP [ (fractional_seconds_precision) ] WITH TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6. For example, you specify TIMESTAMP WITH TIME ZONE as a literal as follows:

```
TIMESTAMP '1997-01-31 09:26:56.66 +02:00'
```

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data. For example,

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '1999-04-15 11:00:00 -5:00'
```

That is, 8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

You can replace the UTC offset with the TZR (time zone region) format element. For example, the following example has the same value as the preceding example:

```
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
```

To eliminate the ambiguity of boundary cases when the daylight savings time switches, use both the TZR and a corresponding TZD format element. The following example ensures that the preceding example will return a daylight savings time value:

```
TIMESTAMP '1999-10-29 01:30:00 US/Pacific PDT'
```

If you do not add the TZD format element, and the datetime value is ambiguous, then Oracle returns an error if you have the ERROR_ON_OVERLAP_TIME session parameter set to TRUE. If that parameter is set to FALSE, then Oracle interprets the ambiguous datetime as standard time.

Note: Oracle's time zone data is derived from the public domain information available at ftp://elsie.nci.nih.gov/pub/. Oracle's time zone data may not reflect the most recent data available at this site. Please refer to *Oracle9i Database Globalization Support Guide* for more information on Oracle time zone data.

See Also:

- "Support for Daylight Savings Times" on page 2-25 and Table 2–15, " Datetime Format Elements" on page 2-70 for information on daylight savings support
- TO_TIMESTAMP_TZ on page 6-183 for information on converting character data to TIMESTAMP WITH TIME ZONE data
- ALTER SESSION on page 10-2 for information on the ERROR_ON_OVERLAP_TIME session parameter

TIMESTAMP WITH LOCAL TIME ZONE Datatype

TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP that includes a **time zone displacement** in its value. It differs from TIMESTAMP WITH

TIME ZONE in that data stored in the database is normalized to the database time zone, and the time zone displacement is not stored as part of the column data. When users retrieve the data, Oracle returns it in the users' local session time zone. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). Specify the TIMESTAMP WITH LOCAL TIME ZONE datatype as follows:

```
TIMESTAMP [ (fractional_seconds_precision) ] WITH LOCAL TIME ZONE
```

where *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

There is no literal for TIMESTAMP WITH LOCAL TIME ZONE.

Note: Oracle's time zone data is derived from the public domain information available at ftp://elsie.nci.nih.gov/pub/. Oracle's time zone data may not reflect the most recent data available at this site. Please refer to *Oracle9i Database Globalization Support Guide* for more information on Oracle time zone data.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for examples of using this datatype
- CAST on page 6-25 for information on converting character data to TIMESTAMP WITH LOCAL TIME ZONE

INTERVAL YEAR TO MONTH Datatype

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where *year_precision* is the number of digits in the YEAR datetime field. The default value of *year_precision* is 2.

> **Note:** You have a great deal of flexibility when specifying interval values as literals. Please refer to "Interval Literals" on page 2-57 for detailed information on specify interval values as literals.

INTERVAL DAY TO SECOND Datatype

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. Specify this datatype as follows:

```
INTERVAL DAY [(day_precision)]
   TO SECOND [(fractional_seconds_precision)]
```

where

- *day_precision* is the number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2.
- *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

> **Note:** You have a great deal of flexibility when specifying interval values as literals. Please refer to "Interval Literals" on page 2-57 for detailed information on specify interval values as literals.

Datetime/Interval Arithmetic

Oracle lets you derive datetime and interval value expressions. Datetime value expressions yield values of datetime datatype. Interval value expressions yield values of interval datatype. Table 2–5 lists the operators that you can use in these expressions.

Table 2–5 Operators in Datetime/Interval Value Expressions

Operand 1	Operator	Operand 2	Result Type
Datetime	+	Interval	Datetime
Datetime	-	Interval	Datetime
Interval	+	Datetime	Datetime
Datetime	-	Datetime	Interval
Interval	+	Interval	Interval

Table 2–5 Operators in Datetime/Interval Value Expressions

Operand 1	Operator	Operand 2	Result Type
Interval	-	Interval	Interval
Interval	*	Numeric	Interval
Numeric	*	Interval	Interval
Interval	/	Numeric	Interval

For example, you can add an interval value expression to a start time. Consider the sample table `oe.orders` with a column `order_date`. The following statement adds 30 days to the value of the `order_date` column:

```
SELECT order_id, order_date + INTERVAL '30' DAY FROM orders;
```

Oracle performs all timestamp arithmetic in UTC time. For `TIMESTAMP WITH LOCAL TIME ZONE`, Oracle converts the datetime value from the database time zone to UTC and converts back to the database time zone after performing the arithmetic. For `TIMESTAMP WITH TIME ZONE`, the datetime value is always in UTC, so no conversion is necessary.

Support for Daylight Savings Times

Oracle automatically determines, for any given time zone region, whether daylight savings is in effect and returns local time values based accordingly. The datetime value is sufficient for Oracle to determine whether daylight savings time is in effect for a given region in all cases except **boundary cases**. A boundary case occurs during the period when daylight savings goes into or comes out of effect. For example, in the US-Pacific region, when daylight savings goes into effect, the time changes from 2:00 a.m. to 3:00 a.m. The one hour interval between 2 and 3 a.m. does not exist. When daylight savings goes out of effect, the time changes from 2:00 a.m. back to 1:00 a.m., and the one-hour interval between 1 and 2 a.m. is repeated.

To resolve these boundary cases, Oracle uses the `TZR` and `TZD` format elements, as described in Table 2–15 on page 2-70. `TZR` represents the time zone region in datetime input strings. Examples are '`Australia/North`', '`UTC`', and '`Singapore`'. `TZD` represents an abbreviated form of the time zone region with daylight savings information. Examples are '`PST`' for US/Pacific standard time and '`PDT`' for US/Pacific daylight time. To see a listing of valid values for the `TZR` and `TZD` format elements, query the `TZNAME` and `TZABBREV` columns of the `V$TIMEZONE_NAMES` dynamic performance view.

Note: Timezone region names are needed by the daylight savings feature. The region names are stored in two time zone files. The default time zone file is a small file containing only the most common time zones to maximize performance. If your time zone is not in the default file, then you will not have daylight savings support until you provide a path to the complete (larger) file by way of the `ORA_TZFILE` environment variable. Please refer to *Oracle9i Database Administrator's Guide* for more information about setting the `ORA_TZFILE` environment variable.

Note: Oracle's time zone data is derived from the public domain information available at ftp://elsie.nci.nih.gov/pub/. Oracle's time zone data may not reflect the most recent data available at this site. Please refer to *Oracle9i Database Globalization Support Guide* for more information on Oracle time zone data.

See Also:

- "Date Format Models" on page 2-68 for information on the format elements
- *Oracle9i Database Reference* for information on the dynamic performance views

Datetime and Interval Example

The following example shows how to declare some datetime and interval datatypes.

```
CREATE TABLE my_table (
   start_time      TIMESTAMP,
   duration_1      INTERVAL DAY (6) TO SECOND (5),
   duration_2      INTERVAL YEAR TO MONTH);
```

The `start_time` column is of type `TIMESTAMP`. The implicit fractional seconds precision of `TIMESTAMP` is 6.

The `duration_1` column is of type `INTERVAL DAY TO SECOND`. The maximum number of digits in field `DAY` is 6 and the maximum number of digits in the fractional second is 5. (The maximum number of digits in all other datetime fields is 2.)

The duration_2 column is of type INTERVAL YEAR TO MONTH. The maximum number of digits of the value in each field (YEAR and MONTH) is 2.

RAW and LONG RAW Datatypes

The RAW and LONG RAW datatypes store data that is not to be interpreted (not explicitly converted when moving data between different systems) by Oracle. These datatypes are intended for binary data or byte strings. For example, you can use LONG RAW to store graphics, sound, documents, or arrays of binary data, for which the interpretation is dependent on the use.

Note: Oracle Corporation strongly recommends that you convert LONG RAW columns to binary LOB (BLOB) columns. LOB columns are subject to far fewer restrictions than LONG columns. See TO_LOB on page 6-175 for more information.

RAW is a variable-length datatype like VARCHAR2, except that Oracle Net (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Oracle Net and Import/Export automatically convert CHAR, VARCHAR2, and LONG data from the database character set to the user session character set (which you can set with the NLS_LANGUAGE parameter of the ALTER SESSION statement), if the two character sets are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form, with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

Large Object (LOB) Datatypes

The built-in LOB datatypes BLOB, CLOB, and NCLOB (stored internally) and BFILE (stored externally), can store large and unstructured data such as text, image, video, and spatial data up to 4 gigabytes in size.

When creating a table, you can optionally specify different tablespace and storage characteristics for LOB columns or LOB object attributes from those specified for the table.

LOB columns contain LOB locators that can refer to out-of-line or in-line LOB values. Selecting a LOB from a table actually returns the LOB's locator and not the

entire LOB value. The DBMS_LOB package and Oracle Call Interface (OCI) operations on LOBs are performed through these locators.

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

- LOBs can be attributes of a user-defined datatype (object).
- The LOB locator is stored in the table column, either with or without the actual LOB value. BLOB, NCLOB, and CLOB values can be stored in separate tablespaces. BFILE data is stored in an external file on the server.
- When you access a LOB column, the locator is returned.
- A LOB can be up to 4 gigabytes in size. BFILE maximum size is operating system dependent, but cannot exceed 4 gigabytes.
- LOBs permit efficient, random, piece-wise access to and manipulation of data.
- You can define more than one LOB column in a table.
- With the exception of NCLOB, you can define one or more LOB attributes in an object.
- You can declare LOB bind variables.
- You can select LOB columns and LOB attributes.
- You can insert a new row or update an existing row that contains one or more LOB columns or an object with one or more LOB attributes. (You can set the internal LOB value to NULL, empty, or replace the entire LOB with data. You can set the BFILE to NULL or make it point to a different file.)
- You can update a LOB row / column intersection or a LOB attribute with another LOB row / column intersection or LOB attribute.
- You can delete a row containing a LOB column or LOB attribute and thereby also delete the LOB value. Note that for BFILEs, the actual operating system file is not deleted.

You can access and populate rows of an internal LOB column (a LOB column stored in the database) simply by issuing an INSERT or UPDATE statement. However, to access and populate a LOB attribute that is part of an object type, you must first initialize the LOB attribute using the EMPTY_CLOB or EMPTY_BLOB function. You can then select the empty LOB attribute and populate it using the DBMS_LOB package or some other appropriate interface.

Restrictions on LOB Columns

LOB columns are subject to the following restrictions:

- Distributed LOBs are not supported. Therefore, you cannot use a remote locator in SELECT or WHERE clauses of queries or in functions of the DBMS_LOB package.

  The following syntax is not supported for LOBs:

  ```
  SELECT lobcol FROM table1@remote_site;
  INSERT INTO lobtable SELECT type1.lobattr FROM table1@remote_
  site;
  SELECT DBMS_LOB.getlength(lobcol) FROM table1@remote_site;
  ```

  However, you can use a remote locator in others parts of queries that reference LOBs. The following syntax is supported on remote LOB columns:

  ```
  CREATE TABLE t AS SELECT * FROM table1@remote_site;
  INSERT INTO t SELECT * FROM table1@remote_site;
  UPDATE t SET lobcol = (SELECT lobcol FROM table1@remote_site);
  INSERT INTO table1@remote_site ...
  UPDATE table1@remote_site ...
  DELETE table1@remote_site ...
  ```

  For the first three types of statement, which contain subqueries, only standalone LOB columns are allowed in the select list. SQL functions or DBMS_LOB APIs on LOBs are not supported. For example, the following statement is supported:

  ```
  CREATE TABLE AS SELECT clob_col FROM tab@dbs2;
  ```

  However, the following statement is not supported:

  ```
  CREATE TABLE AS SELECT dbms_lob.substr(clob_col) from tab@dbs2;
  ```

- Clusters cannot contain LOBs, either as key or nonkey columns.
- You cannot create a varray of LOBs.
- You cannot specify LOB columns in the ORDER BY clause of a query, or in the GROUP BY clause of a query or in an aggregate function.
- You cannot specify a LOB column in a SELECT ... DISTINCT or SELECT ... UNIQUE statement or in a join. However, you can specify a LOB attribute of an object type column in a SELECT ... DISTINCT statement or in a query that uses the UNION or MINUS set operator if the column's object type has a MAP or ORDER function defined on it.
- You cannot specify LOB columns in ANALYZE ... COMPUTE or ANALYZE ... ESTIMATE statements.

- The first (INITIAL) extent of a LOB segment must contain at least three database blocks.
- When creating an UPDATE DML trigger, you cannot specify a LOB column in the UPDATE OF clause.
- You cannot specify a LOB as a primary key column.
- You cannot specify a LOB column as part of an index key. However, you can specify a LOB column in the function of a function-based index or in the indextype specification of a domain index. In addition, Oracle Text lets you define an index on a CLOB column.

  See Also: *Oracle9i Data Cartridge Developer's Guide* for more information about defining triggers on domain indexes

- In an INSERT or UPDATE operation, you can bind data of any size to a LOB column, but you cannot bind data to a LOB attribute of an object type. In an INSERT ... AS SELECT operation, you can bind up to 4000 bytes of data to LOB columns.

  See Also: "Keywords and Parameters" section of individual SQL statements in *Oracle9i SQL Reference* for additional semantics for the use of LOBs

- If a table has both LONG and LOB columns, you cannot bind more than 4000 bytes of data to both the LONG and LOB columns in the same SQL statement. However, you can bind more than 4000 bytes of data to either the LONG or the LOB column.

Notes:

- Oracle8*i* Release 2 (8.1.6) and higher support the `CACHE READS` setting for LOBs. If you have such LOBs and you downgrade to an earlier release, Oracle generates a warning and converts the LOBs from `CACHE READS` to `CACHE LOGGING`. You can subsequently alter the LOBs to either `NOCACHE LOGGING` or `NOCACHE NOLOGGING`. For more information see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*
- For a table on which you have defined a DML trigger, if you use OCI functions or `DBMS_LOB` routines to change the value of a LOB column or the LOB attribute of an object type column, Oracle does not fire the DML trigger.

See Also:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for more information about LOBs, including details about LOB restrictions
- EMPTY_BLOB, EMPTY_CLOB on page 6-59
- "Oracle-Supplied Types" on page 2-40 for alternative ways of storing image, audio, video, and spatial data

The following example shows how the sample table `pm.print_media` was created. (This example assumes the existence of the `textdoc_tab` object table, which is nested table in the `print_media` table.)

```
CREATE TABLE print_media
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    , press_release     LONG
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab;
```

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* and *Oracle Call Interface Programmer's Guide* for more information about these interfaces and LOBs
- the *modify_column_options* clause of ALTER TABLE on page 11-2 and TO_LOB on page 6-175 for more information on converting LONG columns to LOB columns

BFILE Datatype

The BFILE datatype enables access to binary file LOBs that are stored in file systems outside the Oracle database. A BFILE column or attribute stores a BFILE locator, which serves as a pointer to a binary file on the server's file system. The locator maintains the directory alias and the filename.

You can change the filename and path of a BFILE without affecting the base table by using the BFILENAME function.

See Also: BFILENAME on page 6-22 for more information on this built-in SQL function

Binary file LOBs do not participate in transactions and are not recoverable. Rather, the underlying operating system provides file integrity and durability. The maximum file size supported is 4 gigabytes.

The database administrator must ensure that the file exists and that Oracle processes have operating system read permissions on the file.

The BFILE datatype enables read-only support of large binary files. You cannot modify or replicate such a file. Oracle provides APIs to access file data. The primary interfaces that you use to access file data are the DBMS_LOB package and the OCI.

See Also:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* and *Oracle Call Interface Programmer's Guide* for more information about LOBs.
- CREATE DIRECTORY on page 13-46

BLOB Datatype

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bitstreams with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. BLOB value manipulations can be committed and rolled back. However, you cannot save a BLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

CLOB Datatype

The CLOB datatype stores single-byte and multibyte character data. Both fixed-width and variable-width character sets are supported, and both use the CHAR database character set. CLOBs can store up to 4 gigabytes of character data.

CLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. CLOB value manipulations can be committed and rolled back. However, you cannot save a CLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB Datatype

The NCLOB datatype stores Unicode data using the national character set. Both fixed-width and variable-width character sets are supported. NCLOBs can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. NCLOB value manipulations can be committed and rolled back. However, you cannot save an NCLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

> **See Also:** *Oracle9i Database Globalization Support Guide* for information on Unicode datatype support

ROWID Datatype

Each row in the database has an address. You can examine a row's address by querying the pseudocolumn ROWID. Values of this pseudocolumn are strings representing the address of each row. These strings have the datatype ROWID. You can also create tables and clusters that contain actual columns having the ROWID

datatype. Oracle does not guarantee that the values of such columns are valid rowids.

> **See Also:** "Pseudocolumns" on page 2-83 for more information on the ROWID pseudocolumn

Restricted Rowids

Beginning with Oracle8, Oracle SQL incorporated an extended format for rowids to efficiently support partitioned tables and indexes and tablespace-relative data block addresses (DBAs) without ambiguity.

Character values representing rowids in Oracle7 and earlier releases are called **restricted** rowids. Their format is as follows:

```
block.row.file
```

where:

- *block* is a hexadecimal string identifying the data block of the datafile containing the row. The length of this string depends on your operating system.
- *row* is a four-digit hexadecimal string identifying the row in the data block. The first row of the block has a digit of 0.
- *file* is a hexadecimal string identifying the database file containing the row. The first datafile has the number 1. The length of this string depends on your operating system.

Extended Rowids

The **extended** ROWID datatype stored in a user column includes the data in the restricted rowid plus a **data object number**. The data object number is an identification number assigned to every database segment. You can retrieve the data object number from the data dictionary views USER_OBJECTS, DBA_OBJECTS, and ALL_OBJECTS. Objects that share the same segment (clustered tables in the same cluster, for example) have the same object number.

Extended rowids are stored as base 64 values that can contain the characters A-Z, a-z, 0-9, as well as the plus sign (+) and forward slash (/). Extended rowids are not available directly. You can use a supplied package, DBMS_ROWID, to interpret extended rowid contents. The package functions extract and provide information that would be available directly from a restricted rowid as well as information specific to extended rowids.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the functions available with the DBMS_ROWID package and how to use them

Compatibility and Migration

The restricted form of a rowid is still supported in Oracle9*i* for backward compatibility, but all tables return rowids in the extended format.

See Also: *Oracle9i Database Migration* for information regarding compatibility and migration issues

UROWID Datatype

Each row in a database has an address. However, the rows of some tables have addresses that are not physical or permanent or were not generated by Oracle. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Rowids of foreign tables (such as DB2 tables accessed through a gateway) are not standard Oracle rowids.

Oracle uses "universal rowids" (**urowids**) to store the addresses of index-organized and foreign tables. Index-organized tables have logical urowids and foreign tables have foreign urowids. Both types of urowid are stored in the ROWID pseudocolumn (as are the physical rowids of heap-organized tables).

Oracle creates logical rowids based on a table's primary key. The logical rowids do not change as long as the primary key does not change. The ROWID pseudocolumn of an index-organized table has a datatype of UROWID. You can access this pseudocolumn as you would the ROWID pseudocolumn of a heap-organized table (that is, using the SELECT ROWID statement). If you wish to store the rowids of an index-organized table, then you can define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

Note: Heap-organized tables have physical rowids. Oracle Corporation does not recommend that you specify a column of datatype UROWID for a heap-organized table.

See Also:

- *Oracle9i Database Concepts* and *Oracle9i Database Performance Tuning Guide and Reference* for more information on the UROWID datatype and how Oracle generates and manipulates universal rowids
- "ROWID Datatype" on page 2-33 for a discussion of the address of database rows

ANSI, DB2, and SQL/DS Datatypes

SQL statements that create tables and clusters can also use ANSI datatypes and datatypes from IBM's products SQL/DS and DB2. Oracle recognizes the ANSI or IBM datatype name that differs from the Oracle datatype name, records it as the name of the datatype of the column, and then stores the column's data in an Oracle datatype based on the conversions shown in Table 2–6 and Table 2–7.

Table 2–6 ANSI Datatypes Converted to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER(n) CHAR(n)	CHAR(n)
CHARACTER VARYING(n) CHAR VARYING(n)	VARCHAR(n)
NATIONAL CHARACTER(n) NATIONAL CHAR(n) NCHAR(n)	NCHAR(n)
NATIONAL CHARACTER VARYING(n) NATIONAL CHAR VARYING(n) NCHAR VARYING(n)	NVARCHAR2(n)
NUMERIC(p,s) DECIMAL(p,s)[a]	NUMBER(p,s)

[a]The NUMERIC and DECIMAL datatypes can specify only fixed-point numbers. For these datatypes, s defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision b. The default precision for this datatype is 126 binary, or 38 decimal.

[c]The DOUBLE PRECISION datatype is a floating-point number with binary precision 126.

[d]The REAL datatype is a floating-point number with a binary precision of 63, or 18 decimal.

Table 2–6 (Cont.) ***ANSI Datatypes Converted to Oracle Datatypes***

ANSI SQL Datatype	Oracle Datatype
INTEGER	NUMBER(38)
INT	
SMALLINT	
FLOAT(b)[b]	NUMBER
DOUBLE PRECISION[c]	
REAL[d]	

[a]The NUMERIC and DECIMAL datatypes can specify only fixed-point numbers. For these datatypes, s defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision b. The default precision for this datatype is 126 binary, or 38 decimal.

[c]The DOUBLE PRECISION datatype is a floating-point number with binary precision 126.

[d]The REAL datatype is a floating-point number with a binary precision of 63, or 18 decimal.

Table 2–7 SQL/DS and DB2 Datatypes Converted to Oracle Datatypes

SQL/DS or DB2 Datatype	Oracle Datatype
CHARACTER(n)	CHAR(n)
VARCHAR(n)	VARCHAR(n)
LONG VARCHAR(n)	LONG
DECIMAL(p,s)[a]	NUMBER(p,s)
INTEGER	NUMBER(38)
SMALLINT	
FLOAT(b)b	NUMBER

[a]The DECIMAL datatype can specify only fixed-point numbers. For this datatype, *s* defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision *b*. The default precision for this datatype is 126 binary, or 38 decimal.

Do not define columns with the following SQL/DS and DB2 datatypes, because they have no corresponding Oracle datatype:

- GRAPHIC
- LONG VARGRAPHIC

- VARGRAPHIC
- TIME

Note that data of type TIME can also be expressed as Oracle DATE data.

User-Defined Types

User-defined datatypes use Oracle built-in datatypes and other user-defined datatypes as the building blocks of types that model the structure and behavior of data in applications.

The sections that follow describe the various categories of user-defined types.

See Also:

- *Oracle9i Database Concepts* for information about Oracle built-in datatypes
- CREATE TYPE on page 16-3 and the CREATE TYPE BODY on page 16-25 for information about creating user-defined types
- *Oracle9i Application Developer's Guide - Fundamentals* for information about using user-defined types

Object Types

Object types are abstractions of the real-world entities, such as purchase orders, that application programs deal with. An object type is a schema object with three kinds of components:

- A **name**, which identifies the object type uniquely within that schema
- **Attributes**, which are built-in types or other user-defined types. Attributes model the structure of the real-world entity.
- **Methods**, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C or Java and stored externally. Methods implement operations the application can perform on the real-world entity.

REFs

An **object identifier** (OID) uniquely identifies an object and enables you to reference the object from other objects or from relational tables. A datatype category called REF represents such references. A REF is a container for an object identifier. REFs are pointers to objects.

When a REF value points to a nonexistent object, the REF is said to be "dangling". A dangling REF is different from a null REF. To determine whether a REF is dangling or not, use the predicate IS [NOT] DANGLING. For example, given object view oc_orders in the sample schema oe, the column customer_ref is of type REF to type customer_typ, which has an attribute cust_email:

```
SELECT o.customer_ref.cust_email
   FROM oc_orders o
   WHERE o.customer_ref IS NOT DANGLING;
```

Varrays

An array is an ordered set of data elements. All elements of a given array are of the same datatype. Each element has an **index**, which is a number corresponding to the element's position in the array.

The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called **varrays**. You must specify a maximum size when you declare the array.

When you declare a varray, it does not allocate space. It defines a type, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

Oracle normally stores an array object either in line (that is, as part of the row data) or out of line (in a LOB), depending on its size. However, if you specify separate storage characteristics for a varray, then Oracle will store it out of line, regardless of its size.

> **See Also:** the *varray_col_properties* of CREATE TABLE on page 15-39

Nested Tables

A nested table type models an unordered set of elements. The elements may be built-in types or user-defined types. You can view a nested table as a single-column table or, if the nested table is an object type, as a multicolumn table, with a column for each attribute of the object type.

A nested table definition does not allocate space. It defines a type, which you can use to declare:

- Columns of a relational table
- Object type attributes
- PL/SQL variables, parameters, and function return values

When a nested table appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

Oracle-Supplied Types

Oracle Corporation provides SQL-based interfaces for defining new types when the built-in or ANSI-supported types are not sufficient. The behavior for these types can be implemented in C/C++, Java, or PL/ SQL. Oracle automatically provides the low-level infrastructure services needed for input-output, heterogeneous client-side access for new datatypes, and optimizations for data transfers between the application and the database.

These interfaces can be used to build user-defined (or object) types, and are also used by Oracle to create some commonly useful datatypes. Several such datatypes are supplied with the server, and they serve both broad horizontal application areas (for example, the "Any" types) and specific vertical ones (for example, the spatial type).

The Oracle-supplied types, along with cross-references to the documentation of their implementation and use, are described in the following sections:

- "Any" Types
- XML Types
- Spatial Type
- Media Types

"Any" Types

The "Any" types provide highly flexible modeling of procedure parameters and table columns where the actual type is not known. These datatypes let you dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type. These types have OCI and PL/SQL interfaces for construction and access.

SYS.ANYTYPE

This type can contain a type description of any named SQL type or unnamed transient type.

SYS.ANYDATA

This type contains an instance of a given type, with data, plus a description of the type. ANYDATA can be used as a table column datatype and lets you store heterogeneous values in a single column. The values can be of SQL built-in types as well as user-defined types.

SYS.ANYDATASET

This type contains a description of a given type plus a set of data instances of that type. ANYDATASET can be used as a procedure parameter datatype where such flexibility is needed. The values of the data instances can be of SQL built-in types as well as user-defined types.

> **See Also:** *Oracle Call Interface Programmer's Guide*, *PL/SQL User's Guide and Reference*, and *Oracle9i Supplied PL/SQL Packages and Types Reference* for the implementation of these types and guidelines for using them

XML Types

Extensible Markup Language (XML) is a standard format developed by the World Wide Web Consortium (W3C) for representing structured and unstructured data on the Web. Universal Resource Identifiers (URIs) identify resources such as Web pages anywhere on the Web. Oracle provides types to handle XML and URI data, as well as a class of URIs called DBURI-REFs to access data stored within the database itself. It also provides a new set of types to store and access both external and internal URIs from within the database.

XMLType

This Oracle-supplied type can be used to store and query XML data in the database. XMLType has member functions you can use to access, extract, and query the XML data using XPath expressions. XPath is another standard developed by the W3C committee to traverse XML documents. Oracle XMLType functions support many W3C XPath expressions. Oracle also provides a set of SQL functions and PL/SQL packages to create XMLType values from existing relational or object-relational data.

XMLType is a system-defined type, so you can use it as an argument of a function or as the datatype of a table or view column. You can also create tables and views of XMLType. When you create an XMLType column in a table, you can choose to store the XML data in a CLOB column or object relationally.

You can also register the schema (using the DBMS_XMLSCHEMA package) and create a table or column conforming to the registered schema. In this case Oracle stores the XML data in underlying object-relational columns by default, but you can specify storage in a CLOB column even for schema-based data.

Queries and DML on XMLType columns operate the same regardless of the storage mechanism.

URI Datatypes

Oracle supplies a family of URI types—URIType, DBURIType, XDBURIType, and HTTPURIType—which are related by an inheritance hierarchy. URIType is an object type and the others are subtypes of URIType. Since URIType is the supertype, you can create columns of this type and store DBURIType or HTTPURIType type instances in this column.

HTTPURIType You can use HTTPURIType to store URLs to external Web pages or to files. Oracle accesses these files using the HTTP (Hypertext Transfer Protocol) protocol.

XDBURIType You can use XDBURIType to expose documents in the XML database hierarchy as URIs that can be embedded in any URIType column in a table. The XDBURIType consists of a URL, which comprises the hierarchical name of the XML document to which it refers and an optional fragment representing the XPath syntax. The fragment is separated from the URL part by a pound sign (#). The following lines are examples of XDBURIType:

```
/home/oe/doc1.xml
/home/oe/doc1.xml#/orders/order_item
```

DBURIType DBURIType can be used to store DBURI-REFs, which reference data inside the database. Storing DBURI-REFs lets you reference data stored inside or outside the database and access the data consistently.

DBURI-REFs use an XPath-like representation to reference data inside the database. If you imagine the database as an XML tree, then you would see the tables, rows, and columns as elements in the XML document. For instance, the sample human resources user hr would see the following XML tree:

```
<HR>
  <EMPLOYEES>
    <ROW>
      <EMPLOYEE_ID>205</EMPLOYEE_ID>
      <LAST_NAME>Higgins</LAST_NAME>
      <SALARY>12000</SALARY>
      .. <!-- other columns -->
    </ROW>
    ... <!-- other rows -->
  </EMPLOYEES>
  <!-- other tables..-->
</HR>
<!-- other user schemas on which you have some privilege on..-->
```

The DBURI-REF is simply an XPath expression over this virtual XML document. So to reference the SALARY value in the EMPLOYEES table for the employee with employee number 205, we can write a DBURI-REF as,

```
/HR/EMPLOYEES/ROW[EMPLOYEE_ID=205]/SALARY
```

Using this model, you can reference data stored in CLOB columns or other columns and expose them as URLs to the external world.

URIFactory Package

Oracle also provides the URIFactory package, which can create and return instances of the various subtypes of the URITypes. The package analyzes the URL string, identifies the type of URL (HTTP, DBURI, and so on), and creates an instance of the subtype. To create a DBURI instance, the URL must start with the prefix /oradb. For example, UriFactory.getUri('/oradb/HR/EMPLOYEES') would create a DBUriType instance and UriFactory.getUri('/sys/schema') would create an XDBUriType instance.

See Also:

- *Oracle9i Application Developer's Guide - Object-Relational Features* for general information on object types and type inheritance
- *Oracle9i XML Developer's Kits Guide - XDK* for more information about these supplied types and their implementation
- *Oracle9i Application Developer's Guide - Advanced Queuing* for information about using XMLType with Oracle Advanced Queuing

Spatial Type

The object-relational implementation of Oracle Spatial consists of a set of object data types, an index method type, and operators on these types.

MDSYS.SDO_GEOMETRY

The geometric description of a spatial object is stored in a single row, in a single column of object type `SDO_GEOMETRY` in a user-defined table. Any table that has a column of type `SDO_GEOMETRY` must have another column, or set of columns, that defines a unique primary key for that table. Tables of this sort are sometimes referred to as geometry tables.

> **See Also:** *Oracle Spatial User's Guide and Reference* for information on the implementation of this type and guidelines for using it

Media Types

Oracle interMedia uses object types, similar to Java or C++ classes, to describe multimedia data. An instance of these object types consists of attributes, including metadata and the media data, and methods. The Oracle interMedia types are:

ORDSYS.ORDAudio

The `ORDAUDIO` object type supports the storage and management of audio data.

ORDSYS.ORDImage

The `ORDIMAGE` object type supports the storage and management of image data.

ORDSYS.ORDVideo

The `ORDVIDEO` object type supports the storage and management of video data.

ORDSYS.ORDDoc

The `ORDDOC` object type supports storage and management of any type of media data, including audio, image and video data. Use this type when you want all media to be stored in a single column.

> **See Also:** *Oracle interMedia User's Guide and Reference* for information on the implementation of these types and guidelines for using them

Datatype Comparison Rules

This section describes how Oracle compares values of each datatype.

Number Values

A larger value is considered greater than a smaller one. All negative numbers are less than zero and all positive numbers. Thus, -1 is less than 100; -100 is less than -1.

Date Values

A later date is considered greater than an earlier one. For example, the date equivalent of '29-MAR-1997' is less than that of '05-JAN-1998' and '05-JAN-1998 1:35pm' is greater than '05-JAN-1998 10:09am'.

Character String Values

Character values are compared using one of these comparison rules:

- Blank-padded comparison semantics
- Nonpadded comparison semantics

The following sections explain these comparison semantics.

Blank-Padded Comparison Semantics If the two values have different lengths, then Oracle first adds blanks to the end of the shorter one so their lengths are equal. Oracle then compares the values character by character up to the first character that differs. The value with the greater character in the first differing position is considered greater. If two values have no differing characters, then they are considered equal. This rule means that two values are equal if they differ only in the number of trailing blanks. Oracle uses blank-padded comparison semantics only when both values in the comparison are either expressions of datatype CHAR, NCHAR, text literals, or values returned by the USER function.

Nonpadded Comparison Semantics Oracle compares two values character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values of different length are identical up to the end of the shorter one, then the longer value is considered greater. If two values of equal length have no differing characters, then the values are considered equal. Oracle uses nonpadded comparison semantics whenever one or both values in the comparison have the datatype VARCHAR2 or NVARCHAR2.

The results of comparing two character values using different comparison semantics may vary. The table that follows shows the results of comparing five pairs

of character values using each comparison semantic. Usually, the results of blank-padded and nonpadded comparisons are the same. The last comparison in the table illustrates the differences between the blank-padded and nonpadded comparison semantics.

Blank-Padded	Nonpadded
'ac' > 'ab'	'ac' > 'ab'
'ab' > 'a '	'ab' > 'a '
'ab' > 'a'	'ab' > 'a'
'ab' = 'ab'	'ab' = 'ab'
'a ' = 'a'	**'a ' > 'a'**

Single Characters

Oracle compares single characters according to their numeric values in the database character set. One character is greater than another if it has a greater numeric value than the other in the character set. Oracle considers blanks to be less than any character, which is true in most character sets.

These are some common character sets:

- 7-bit ASCII (American Standard Code for Information Interchange)
- EBCDIC Code (Extended Binary Coded Decimal Interchange Code)
- ISO 8859/1 (International Standards Organization)
- JEUC Japan Extended UNIX

Portions of the ASCII and EBCDIC character sets appear in Table 2–8 and Table 2–9. Note that uppercase and lowercase letters are not equivalent. Also, note that the numeric values for the characters of a character set may not match the linguistic sequence for a particular language.

Table 2–8 ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	32	;	59
!	33	<	60
"	34	=	61
#	35	>	62

Table 2–8 (Cont.) ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
$	36	?	63
%	37	@	64
&	38	A-Z	65-90
’	39	[	91
(	40	\	92
)	41	]	93
*	42	^	94
+	43	_	95
,	44	‘	96
-	45	a-z	97-122
.	46	{	123
/	47	|	124
0-9	48-57	}	125
:	58	~	126

Table 2–9 EBCDIC Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	64	%	108
¢	74	_	109
.	75	>	110
<	76	?	111
(	77	:	122
+	78	#	123
|	79	@	124
&	80	’	125
!	90	=	126
$	91	"	127

Table 2–9 EBCDIC Character Set

Symbol	Decimal value	Symbol	Decimal value
*	92	a-i	129-137
)	93	j-r	145-153
;	94	s-z	162-169
ÿ	95	A-I	193-201
-	96	J-R	209-217
/	97	S-Z	226-233

Object Values

Object values are compared using one of two comparison functions: MAP and ORDER. Both functions compare object type instances, but they are quite different from one another. These functions must be specified as part of the object type.

> **See Also:** CREATE TYPE on page 16-3 and *Oracle9i Application Developer's Guide - Fundamentals* for a description of MAP and ORDER methods and the values they return

Varrays and Nested Tables

You cannot compare varrays and nested tables in Oracle9*i*.

Data Conversion

Generally an expression cannot contain values of different datatypes. For example, an expression cannot multiply 5 by 10 and then add 'JAMES'. However, Oracle supports both implicit and explicit conversion of values from one datatype to another.

Implicit and Explicit Data Conversion

Oracle recommends that you specify explicit conversions rather than rely on implicit or automatic conversions, for these reasons:

- SQL statements are easier to understand when you use explicit datatype conversion functions.
- Automatic datatype conversion can have a negative impact on performance, especially if the datatype of a column value is converted to that of a constant rather than the other way around.

- Implicit conversion depends on the context in which it occurs and may not work the same way in every case. For example, implicit conversion from a date value to a VARCHAR2 value may return an unexpected year depending on the value of the NLS_DATE_FORMAT parameter.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.

Implicit Data Conversion

Oracle automatically converts a value from one datatype to another when such a conversion makes sense. Table 2–10 is a matrix of Oracle implicit conversions. The table shows all possible conversions, without regard to the direction of the conversion or the context in which it is made. The rules governing these details follow the table.

Table 2–10 Implicit Type Conversion Matrix

	CHAR	VARCHAR2	DATE	DATETIME/ INTERVAL	LONG	NUMBER	RAW	ROWID	CLOB	BLOB	NCHAR	NVARCHAR	NCLOB
CHAR	–	X	X	X	X	X	X	–	X	–	X	X	–
VARCHAR2	X	–	X	X	X	X	X	X	X	–	X	X	–
DATE	X	X	–	–	–	–	–	–	–	–	X	X	–
DATETIME/ INTERVAL	X	X	–	–	X	–	–	–	–	–	X	X	–
LONG	X	X	–	X	–	–	X	–	X	–	X	X	X
NUMBER	X	X	–	–	–	–	–	–	–	–	X	X	–
RAW	X	X	–	–	X	–	–	–	–	X	X	X	–
ROWID	X	X	–	–	–	–	–	–	–	–	X	X	–
CLOB	X	X	–	–	X	–	–	–	–	–	–	–	–
BLOB	–	–	–	–	–	–	X	–	–	–	–	–	–

Table 2–10 Implicit Type Conversion Matrix

	CHAR	VARCHAR2	DATE	DATETIME/ INTERVAL	LONG	NUMBER	RAW	ROWID	CLOB	BLOB	NCHAR	NVARCHAR	NCLOB
NCHAR	X	X	X	X	X	X	X	X	–	–	–	X	X
NVARCHAR2	X	X	X	X	X	X	X	X	–	–	X	–	X
NCLOB	–	–	–	–	X	–	–	–	–	–	X	X	–

The following rules govern the direction in which Oracle makes implicit datatype conversions:

- During INSERT and UPDATE operations, Oracle converts the value to the datatype of the affected column.
- During SELECT FROM operations, Oracle converts the data from the column to the type of the target variable.
- When comparing a character value with a NUMBER value, Oracle converts the character data to NUMBER.
- When comparing a character value with a DATE value, Oracle converts the character data to DATE.
- When you use a SQL function or operator with an argument of a datatype other than the one it accepts, Oracle converts the argument to the accepted datatype.
- When making assignments, Oracle converts the value on the right side of the equal sign (=) to the datatype of the target of the assignment on the left side.
- During concatenation operations, Oracle converts from noncharacter datatypes to CHAR or NCHAR.
- During arithmetic operations on and comparisons between character and noncharacter datatypes, Oracle converts from any character datatype to a number, date, or rowid, as appropriate. In arithmetic operations between CHAR/VARCHAR2 and NCHAR/NVARCHAR2, Oracle converts to a number.
- Comparisons between CHAR/VARCHAR2 and NCHAR/NVARCHAR2 types may entail different character sets. The default direction of conversion in such cases is from the database character set to the national character set. Table 2–11 shows the direction of implicit conversions between different character types.

- Most SQL character functions are enabled to accept CLOBs as parameters, and Oracle performs implicit conversions between CLOB and CHAR types. Therefore, functions that are not yet enabled for CLOBs can accept CLOBs through implicit conversion. In such cases, Oracle converts the CLOBs to CHAR or VARCHAR2 before the function is invoked. If the CLOB is larger than 4000 bytes, then Oracle converts only the first 4000 bytes to CHAR.

Table 2–11 Conversion Direction of Different Character Types

	to CHAR	to VARCHAR2	to NCHAR	to NVARCHAR2
from CHAR	--	VARCHAR2	NCHAR	NVARCHAR2
from VARCHAR2	VARCHAR2	--	NVARCHAR2	NVARCHAR2
from NCHAR	NCHAR	NCHAR	--	NVARCHAR2
from NVARCHAR2	NVARCHAR2	NVARCHAR2	NVARCHAR2	--

Implicit Data Conversion Examples

Text Literal Example The text literal '10' has datatype CHAR. Oracle implicitly converts it to the NUMBER datatype if it appears in a numeric expression as in the following statement:

```
SELECT salary + '10'
   FROM employees;
```

Character and Number Values Example When a condition compares a character value and a NUMBER value, Oracle implicitly converts the character value to a NUMBER value, rather than converting the NUMBER value to a character value. In the following statement, Oracle implicitly converts '200' to 200:

```
SELECT last_name
    FROM employees
    WHERE employee_id = '200';
```

Date Example In the following statement, Oracle implicitly converts '03-MAR-97' to a DATE value using the default date format 'DD-MON-YY':

```
SELECT last_name
    FROM employees
    WHERE hire_date = '03-MAR-97';
```

Rowid Example In the following statement, Oracle implicitly converts the text literal 'AAAFYmAAFAAAAFGAAH' to a rowid value:

```
SELECT last_name
    FROM employees
    WHERE ROWID = 'AAAFd1AAFAAAABSAAH';
```

Explicit Data Conversion

You can also explicitly specify datatype conversions using SQL conversion functions. The following table shows SQL functions that explicitly convert a value from one datatype to another.

Table 2–12 Explicit Type Conversion

	to CHAR, VARCHAR2, NCHAR, NVARCHAR2	to NUMBER	to Datetime/ Interval	to RAW	to ROWID	to LONG, LONG RAW	to CLOB, NCLOB, BLOB
from CHAR, VARCHAR2, NCHAR, NVARCHAR2	TO_CHAR (character) TO_NCHAR (character)	TO_ NUMBER	TO_DATE TO_TIMESTAMP TO_TIMESTAMP_TZ TO_YMINTERVAL TO_DSINTERVAL	HEX-TORAW	CHARTO-ROWID	—	TO_CLOB TO_NCLOB
from NUMBER	TO_CHAR (number) TO_NCHAR (number)	—	TO_DATE NUMTOYMINTERVAL NUMTODSINTERVAL	—	—	—	—
from Datetime/ Interval	TO_CHAR (date) TO_NCHAR (datetime)	—	—	—	—	—	—
from RAW	RAWTOHEX RAWTONHEX	—	—	—	—	—	TO_BLOB
from ROWID	ROWIDTOCHAR	—	—	—	—	—	—
from LONG / LONG RAW	—	—	—	—	—	—	TO_LOB
from CLOB, NCLOB, BLOB	TO_CHAR TO_NCHAR	—	—	—	—	—	TO_CLOB TO_NCLOB

Note: You cannot specify LONG and LONG RAW values in cases in which Oracle can perform implicit datatype conversion. For example, LONG and LONG RAW values cannot appear in expressions with functions or operators. For information on the limitations on LONG and LONG RAW datatypes, see "LONG Datatype" on page 2-14.

See Also: "Conversion Functions" on page 6-5 of the SQL Reference for details on all of the explicit conversion functions

Literals

The terms **literal** and **constant value** are synonymous and refer to a fixed data value. For example, 'JACK', 'BLUE ISLAND', and '101' are all character literals; 5001 is a numeric literal. Character literals are enclosed in single quotation marks, which enable Oracle to distinguish them from schema object names.

This section contains these topics:

- Text Literals
- Integer Literals
- Number Literals
- Interval Literals

Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions. You can specify character literals with the '*text*' notation, national character literals with the *N'text'* notation, and numeric literals with the *integer* or *number* notation, depending on the context of the literal. The syntactic forms of these notations appear in the sections that follow.

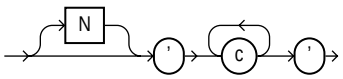
To specify a datetime or interval datatype as a literal, you must take into account any optional precisions included in the datatypes. Examples of specifying datetime and interval datatypes as literals are provided in the relevant sections of "Datatypes" on page 2-2.

Text Literals

Text specifies a text or character literal. You must use this notation to specify values whenever *'text'* or *char* appear in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of text is as follows:

text::=



where

- N specifies representation of the literal using the national character set. Text entered using this notation is translated into the national character set by Oracle when used.
- *c* is any member of the user's character set, except a single quotation mark (').
- '' are two single quotation marks that begin and end text literals. To represent one single quotation mark within a literal, enter two single quotation marks.

A text literal must be enclosed in single quotation marks. This reference uses the terms **text literal** and **character literal** interchangeably.

Text literals have properties of both the CHAR and VARCHAR2 datatypes:

- Within expressions and conditions, Oracle treats text literals as though they have the datatype CHAR by comparing them using blank-padded comparison semantics.
- A text literal can have a maximum length of 4000 bytes.

Here are some valid text literals:

```
'Hello'
'ORACLE.dbs'
'Jackie''s raincoat'
'09-MAR-98'
N'nchar literal'
```

See Also:

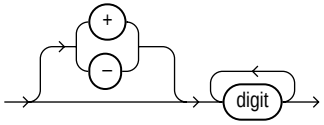
- "About SQL Expressions" on page 4-2 for the syntax description of *expr*
- "Blank-Padded Comparison Semantics" on page 2-45

Integer Literals

You must use the integer notation to specify an integer whenever *integer* appears in expressions, conditions, SQL functions, and SQL statements described in other parts of this reference.

The syntax of *integer* is as follows:

integer::=



where *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

An integer can store a maximum of 38 digits of precision.

Here are some valid integers:

```
7
+255
```

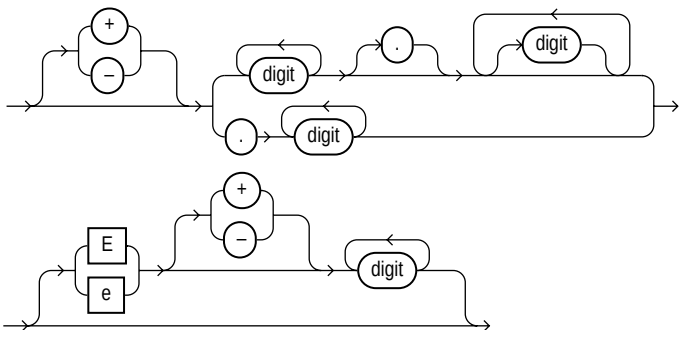
> **See Also:** "About SQL Expressions" on page 4-2 for the syntax description of *expr*

Number Literals

You must use the number notation to specify values whenever *number* appears in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of *number* is as follows:

number::=



where

- + or - indicates a positive or negative value. If you omit the sign, then a positive value is the default.

- *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
- e or E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from -130 to 125.

A *number* can store a maximum of 38 digits of precision.

If you have established a decimal character other than a period (.) with the initialization parameter NLS_NUMERIC_CHARACTERS, then you must specify numeric literals with *'text'* notation. In such cases, Oracle automatically converts the text literal to a numeric value.

For example, if the NLS_NUMERIC_CHARACTERS parameter specifies a decimal character of comma, specify the number 5.123 as follows:

```
'5,123'
```

> **See Also:** ALTER SESSION on page 10-2 and *Oracle9i Database Reference*

Here are some valid representations of *number*:

```
25
+6.34
0.5
25e-03
-1
```

> **See Also:** "About SQL Expressions" on page 4-2 for the syntax description of *expr*

Interval Literals

An interval literal specifies a period of time. You can specify these differences in terms of years and months, or in terms of days, hours, minutes, and seconds. Oracle supports two types of interval literals, YEAR TO MONTH and DAY TO SECOND. Each type contains a leading field and may contain a trailing field. The leading field defines the basic unit of date or time being measured. The trailing field defines the smallest increment of the basic unit being considered. For example, a YEAR TO MONTH interval considers an interval of years to the nearest month. A DAY TO MINUTE interval considers an interval of days to the nearest minute.

If you have date data in numeric form, then you can use the NUMTOYMINTERVAL or NUMTODSINTERVAL conversion function to convert the numeric data into interval literals.

Interval literals are used primarily with analytic functions.

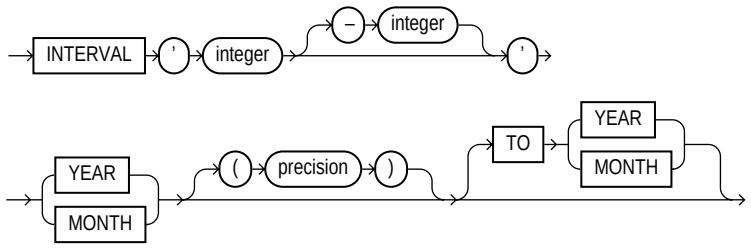
See Also:

- "Analytic Functions" on page 6-9 and *Oracle9i Data Warehousing Guide*
- NUMTODSINTERVAL on page 6-108 and NUMTOYMINTERVAL on page 6-109

INTERVAL YEAR TO MONTH

Specify YEAR TO MONTH interval literals using the following syntax:

interval_year_to_month::=



where

- *'integer [-integer]'* specifies integer values for the leading and optional trailing field of the literal. If the leading field is YEAR and the trailing field is MONTH, then the range of integer values for the month field is 0 to 11.
- *precision* is the maximum number of digits in the leading field. The valid range of the leading field precision is 0 to 9 and its default value is 2.

Restriction: The leading field must be more significant than the trailing field. For example, INTERVAL '0-1' MONTH TO YEAR is not valid.

The following INTERVAL YEAR TO MONTH literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

Examples of the other forms of the literal follow, including some abbreviated versions:

Form of Interval Literal	Interpretation
`INTERVAL '123-2' YEAR(3) TO MONTH`	An interval of 123 years, 2 months. You must specify the leading field precision if it is greater than the default of 2 digits.
`INTERVAL '123' YEAR(3)`	An interval of 123 years 0 months.
`INTERVAL '300' MONTH(3)`	An interval of 300 months.
`INTERVAL '4' YEAR`	Maps to `INTERVAL '4-0' YEAR TO MONTH` and indicates 4 years.
`INTERVAL '50' MONTH`	Maps to `INTERVAL '4-2' YEAR TO MONTH` and indicates 50 months or 4 years 2 months.
`INTERVAL '123' YEAR`	Returns an error, because the default precision is 2, and '123' has 3 digits.

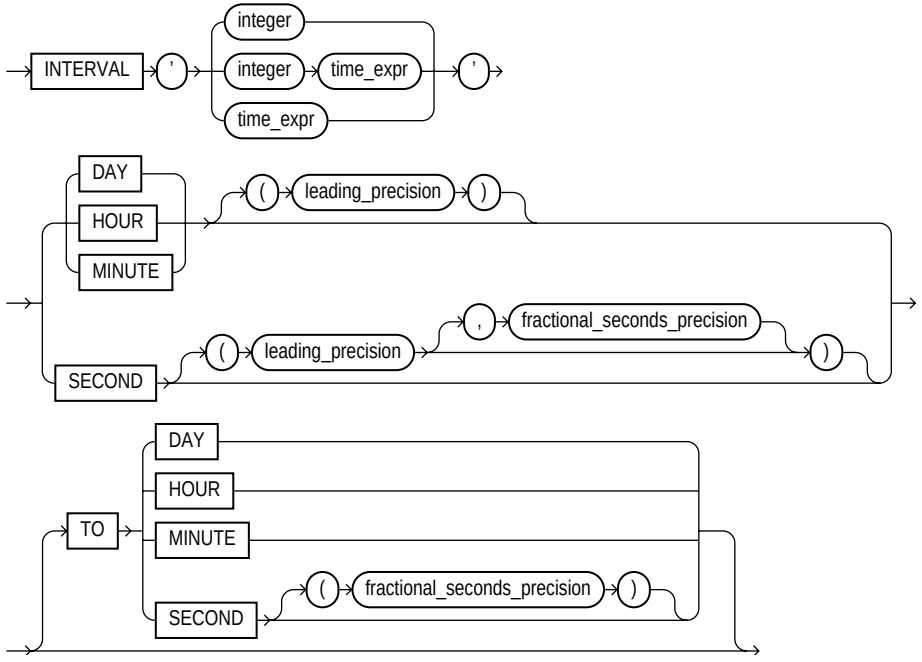
You can add or subtract one `INTERVAL YEAR TO MONTH` literal to or from another to yield another `INTERVAL YEAR TO MONTH` literal. For example:

```
INTERVAL '5-3' YEAR TO MONTH + INTERVAL'20' MONTH =
INTERVAL '6-11' YEAR TO MONTH
```

INTERVAL DAY TO SECOND

Specify `DAY TO SECOND` interval literals using the following syntax:

interval_day_to_second::=



where

- *integer* specifies the number of days. If this value contains more digits than the number specified by the leading precision, then Oracle returns an error.
- *time_expr* specifies a time in the format HH[:MI[:SS[.n]]]or MI[:SS[.n]] or SS[.n], where *n* specifies the fractional part of a second. If *n* contains more digits than the number specified by *fractional_seconds_precision*, then *n* is rounded to the number of digits specified by the *fractional_seconds_precision* value. You can specify *time_expr* following an integer and a space only if the leading field is DAY.
- *leading_precision* is the number of digits in the leading field. Accepted values are 0 to 9. The default is 2.
- *fractional_seconds_precision* is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 1 to 9. The default is 6.

Restriction: The leading field must be more significant than the trailing field. For example, INTERVAL MINUTE TO DAY is not valid. As a result of this restriction, if SECOND is the leading field, the interval literal cannot have any trailing field.

The valid range of values for the trailing field are as follows:

- HOUR: 0 to 23
- MINUTE: 0 to 59
- SECOND: 0 to 59.999999999

Examples of the various forms of INTERVAL DAY TO SECOND literals follow, including some abbreviated versions:

Form of Interval Literal	Interpretation
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)	4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.
INTERVAL '4 5:12' DAY TO MINUTE	4 days, 5 hours and 12 minutes.
INTERVAL '400 5' DAY(3) TO HOUR	400 days 5 hours.
INTERVAL '400' DAY(3)	400 days.
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)	11 hours, 12 minutes, and 10.2222222 seconds.
INTERVAL '11:20' HOUR TO MINUTE	11 hours and 20 minutes.
INTERVAL '10' HOUR	10 hours.
INTERVAL '10:22' MINUTE TO SECOND	10 minutes 22 seconds.
INTERVAL '10' MINUTE	10 minutes.
INTERVAL '4' DAY	4 days.
INTERVAL '25' HOUR	25 hours.
INTERVAL '40' MINUTE	40 minutes.
INTERVAL '120' HOUR(3)	120 hours
INTERVAL '30.12345' SECOND(2,4)	30.1235 seconds. The fractional second '12345' is rounded to '1235' because the precision is 4.

You can add or subtract one DAY TO SECOND interval literal from another DAY TO SECOND literal. For example.

```
INTERVAL'20' DAY - INTERVAL'240' HOUR = INTERVAL'10-0' DAY TO SECOND
```

Format Models

A **format model** is a character literal that describes the format of DATE or NUMBER data stored in a character string. When you convert a character string into a date or number, a format model tells Oracle how to interpret the string. In SQL statements, you can use a format model as an argument of the TO_CHAR and TO_DATE functions:

- To specify the format for Oracle to use to return a value from the database
- To specify the format for a value you have specified for Oracle to store in the database

---

Note: A format model does not change the internal representation of the value in the database.

For example,

- The date format model for the string '17:45:29' is 'HH24:MI:SS'.
- The date format model for the string '11-Nov-1999' is 'DD-Mon-YYYY'.
- The number format model for the string '$2,304.25' is '$9,999.99'.

For lists of date and number format model elements, see Table 2–13, " Number Format Elements" on page 2-65 and Table 2–15, " Datetime Format Elements" on page 2-70.

The values of some formats are determined by the value of initialization parameters. For such formats, you can specify the characters returned by these format elements implicitly using the initialization parameter NLS_TERRITORY. You can change the default date format for your session with the ALTER SESSION statement.

See Also:

- *Oracle9i Database Reference* and *Oracle9i Database Globalization Support Guide* for information on these parameters
- ALTER SESSION on page 10-2 for information on changing the values of these parameters

Format of Return Values: Examples You can use a format model to specify the format for Oracle to use to return values from the database to you.

The following statement selects the salaries of the employees in Department 80 and uses the TO_CHAR function to convert these salaries into character values with the format specified by the number format model '$9,990.99':

```
SELECT last_name employee, TO_CHAR(salary, '$99,990.99')
   FROM employees
   WHERE department_id = 80;
```

Because of this format model, Oracle returns salaries with leading dollar signs, commas every three digits, and two decimal places.

The following statement selects the date on which each employee from Department 20 was hired and uses the TO_CHAR function to convert these dates to character strings with the format specified by the date format model 'fmMonth DD, YYYY':

```
SELECT last_name employee,
    TO_CHAR(hire_date,'fmMonth DD, YYYY') hiredate
    FROM employees
    WHERE department_id = 20;
```

With this format model, Oracle returns the hire dates (as specified by "fm") without blank padding, two digits for the day, and the century included in the year.

> **See Also:** "Format Model Modifiers" on page 2-76 for a description of the fm format element

Supplying the Correct Format Model: Examples When you insert or update a column value, the datatype of the value that you specify must correspond to the column's datatype. You can use format models to specify the format of a value that you are converting from one datatype to another datatype required for a column.

For example, a value that you insert into a DATE column must be a value of the DATE datatype or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE datatype). If the value is in another format, then you must use the TO_DATE function to convert the value to the DATE datatype. You must also use a format model to specify the format of the character string.

The following statement updates Hunold's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '1998 05 20' to a DATE value:

```
UPDATE employees
  SET hire_date = TO_DATE('1998 05 20','YYYY MM DD')
  WHERE last_name = 'Hunold';
```

This remainder of this section describes how to use:

- Number Format Models
- Date Format Models
- Format Model Modifiers

> **See Also:** TO_CHAR (datetime) on page 6-168, TO_CHAR (number) on page 6-170, and TO_DATE on page 6-172

Number Format Models

You can use number format models:

- In the TO_CHAR function to translate a value of NUMBER datatype to VARCHAR2 datatype
- In the TO_NUMBER function to translate a value of CHAR or VARCHAR2 datatype to NUMBER datatype

All number format models cause the number to be rounded to the specified number of significant digits. If a value has more significant digits to the left of the decimal place than are specified in the format, then pound signs (#) replace the value. If a positive value is extremely large and cannot be represented in the specified format, then the infinity sign (~) replaces the value. Likewise, if a negative value is extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~). This event typically occurs when you are using TO_CHAR with a restrictive number format string, causing a rounding operation.

Number Format Elements

A number format model is composed of one or more number format elements. Table 2–13 lists the elements of a number format model. Examples are shown in Table 2–14.

Negative return values automatically contain a leading negative sign and positive values automatically contain a leading space unless the format model contains the MI, S, or PR format element.

Table 2–13 Number Format Elements

Element	Example	Description
, (comma)	`9,999`	Returns a comma in the specified position. You can specify multiple commas in a number format model. **Restrictions:** ■ A comma element cannot begin a number format model. ■ A comma cannot appear to the right of a decimal character or period in a number format model.
. (period)	`99.99`	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** You can specify only one period in a number format model.
$	`$9999`	Returns value with a leading dollar sign.
0	`0999`	Returns leading zeros.
	`9990`	Returns trailing zeros.
9	`9999`	Returns value with the specified number of digits with a leading space if positive or with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
B	`B9999`	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of "0"s in the format model).
C	`C999`	Returns in the specified position the ISO currency symbol (the current value of the `NLS_ISO_CURRENCY` parameter).
D	`99D99`	Returns in the specified position the decimal character, which is the current value of the `NLS_NUMERIC_CHARACTER` parameter. The default is a period (.). **Restriction:** You can specify only one decimal character in a number format model.
EEEE	`9.9EEEE`	Returns a value using in scientific notation.
FM	`FM90.9`	Returns a value with no leading or trailing blanks.

Table 2–13 (Cont.) Number Format Elements

Element	Example	Description
G	`9G999`	Returns in the specified position the group separator (the current value of the `NLS_NUMERIC_CHARACTER` parameter). You can specify multiple group separators in a number format model. **Restriction:** A group separator cannot appear to the right of a decimal character or period in a number format model.
L	`L999`	Returns in the specified position the local currency symbol (the current value of the `NLS_CURRENCY` parameter).
MI	`9999MI`	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing blank. **Restriction:** The MI format element can appear only in the last position of a number format model.
PR	`9999PR`	Returns negative value in <angle brackets>. Returns positive value with a leading and trailing blank. **Restriction:** The PR format element can appear only in the last position of a number format model.
RN	`RN`	Returns a value as Roman numerals in uppercase.
rn	`rn`	Returns a value as Roman numerals in lowercase. Value can be an integer between 1 and 3999.
S	`S9999`	Returns negative value with a leading minus sign (-). Returns positive value with a leading plus sign (+).
	`9999S`	Returns negative value with a trailing minus sign (-). Returns positive value with a trailing plus sign (+). **Restriction:** The S format element can appear only in the first or last position of a number format model.

Table 2–13 (Cont.) Number Format Elements

Element	Example	Description
TM	`TM`	"Text minimum". Returns (in decimal output) the smallest number of characters possible. This element is case-insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If output exceeds 64 characters, then Oracle automatically returns the number in scientific notation. **Restrictions:** ■ You cannot precede this element with any other element. ■ You can follow this element only with 9 or E (only one) or e (only one).
U	`U9999`	Returns in the specified position the "Euro" (or other) dual currency symbol (the current value of the `NLS_DUAL_CURRENCY` parameter).
V	`999V99`	Returns a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the "V".
X	`XXXX` `xxxx`	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, then Oracle rounds it to an integer. **Restrictions:** ■ This element accepts only positive values or 0. Negative values return an error. ■ You can precede this element only with 0 (which returns leading zeroes) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, then the return always has 1 leading blank.

Table 2–14 shows the results of the following query for different values of *number* and *'fmt'*:

```
SELECT TO_CHAR(number, 'fmt')
   FROM DUAL;
```

Table 2–14 Results of Example Number Conversions

number	'fmt'	Result
-1234567890	9999999999S	'1234567890-'
0	99.99	' .00'

Table 2–14 Results of Example Number Conversions (Cont.)

number	'fmt'	Result
+0.1	99.99	' .10'
-0.2	99.99	' -.20'
0	90.99	' 0.00'
+0.1	90.99	' 0.10'
-0.2	90.99	'-0.20'
0	9999	' 0'
1	9999	' 1'
0	B9999	' '
1	B9999	' 1'
0	B90.99	' '
+123.456	999.999	'123.456'
-123.456	999.999	'-123.456'
+123.456	FM999.009	'123.456'
+123.456	9.9EEEE	' 1.2E+02'
+1E+123	9.9EEEE	'1.0E+123'
+123.456	FM9.9EEEE	'1.2E+02'
+123.45	FM999.009	'123.45'
+123.0	FM999.009	'123.00'
+123.45	L999.99	' $123.45'
+123.45	FML999.99	'$123.45'
+1234567890	9999999999S	'1234567890+'

Date Format Models

You can use date format models:

- In the TO_DATE function to translate a character value that is in a format other than the default date format into a DATE value

- In the TO_CHAR function to translate a DATE value that is in a format other than the default date format into a string (for example, to print the date from an application)

The total length of a date format model cannot exceed 22 characters.

The default date format is specified either explicitly with the initialization parameter NLS_DATE_FORMAT or implicitly with the initialization parameter NLS_TERRITORY. You can change the default date format for your session with the ALTER SESSION statement.

See Also:

- *Oracle9i Database Reference* for information on the NLS parameters
- ALTER SESSION on page 10-2

Date Format Elements

A date format model is composed of one or more datetime format elements as listed in Table 2–15 on page 2-70.

- For input format models, format items cannot appear twice, and format items that represent similar information cannot be combined. For example, you cannot use 'SYYYY' and 'BC' in the same format string.
- Some of the datetime format elements cannot be used in the TO_* datetime functions, as noted in Table 2–15.
- The following datetime format elements can be used in interval and timestamp format models, but not in the original DATE format model: FF, TZD, TZH, TZM, and TZR.

Capitalization of Date Format Elements Capitalization in a spelled-out word, abbreviation, or Roman numeral follows capitalization in the corresponding format element. For example, the date format model 'DAY' produces capitalized words like 'MONDAY'; 'Day' produces 'Monday'; and 'day' produces 'monday'.

Punctuation and Character Literals in Date Format Models You can also include these characters in a date format model:

- Punctuation such as hyphens, slashes, commas, periods, and colons
- Character literals, enclosed in double quotation marks

These characters appear in the return value in the same location as they appear in the format model.

Table 2–15 Datetime Format Elements

Element	Specify in TO_* datetime functions?[a]	Meaning
- / , . ; : "text"	Yes	Punctuation and quoted text is reproduced in the result.
AD A.D.	Yes	AD indicator with or without periods.
AM A.M.	Yes	Meridian indicator with or without periods.
BC B.C.	Yes	BC indicator with or without periods.
CC SCC	No	One greater than the first two digits of a four-digit year; "S" prefixes BC dates with "-". For example, '20' from '1900'.
D	Yes	Day of week (1-7).
DAY	Yes	Name of day, padded with blanks to length of 9 characters.
DD	Yes	Day of month (1-31).
DDD	Yes	Day of year (1-366).
DY	Yes	Abbreviated name of day.
E	No	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
EE	No	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).

[a] The TO_* datetime functions are TO_CHAR, TO_DATE, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL, and TO_DSINTERVAL.

Table 2–15 (Cont.) Datetime Format Elements

Element	Specify in TO_* datetime functions?[a]	Meaning
FF [1..9]	Yes	Fractional seconds; no radix character is printed (use the X format element to add the radix character). Use the numbers 1 to 9 after FF to specify the number of digits in the fractional second portion of the datetime value returned. If you do not specify a digit, then Oracle uses the precision specified for the datetime datatype or the datatype's default precision. **Examples**: 'HH:MI:SS.FF' SELECT TO_CHAR(SYSTIMESTAMP, 'SS.FF3') from dual;
HH	Yes	Hour of day (1-12).
HH12	No	Hour of day (1-12).
HH24	Yes	Hour of day (0-23).
IW	No	Week of year (1-52 or 1-53) based on the ISO standard.
IYY IY I	No	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	No	4-digit year based on the ISO standard.
J	Yes	Julian day; the number of days since January 1, 4712 BC. Number specified with 'J' must be integers.
MI	Yes	Minute (0-59).
MM	Yes	Month (01-12; JAN = 01).
MON	Yes	Abbreviated name of month.
MONTH	Yes	Name of month, padded with blanks to length of 9 characters.
PM P.M.	No	Meridian indicator with or without periods.

[a] The TO_* datetime functions are TO_CHAR, TO_DATE, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL, and TO_DSINTERVAL.

Table 2–15 (Cont.) Datetime Format Elements

Element	Specify in TO_* datetime functions?[a]	Meaning
Q	No	Quarter of year (1, 2, 3, 4; JAN-MAR = 1).
RM	Yes	Roman numeral month (I-XII; JAN = I).
RR	Yes	Given a year with 2 digits: ■ If the year is <50 and the last 2 digits of the current year are >=50, then the first 2 digits of the returned year are 1 greater than the first 2 digits of the current year. ■ If the year is >=50 and the last 2 digits of the current year are <50, then the first 2 digits of the returned year are 1 less than the first 2 digits of the current year. **See Also:** Table 2–17 on page 2-76
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you don't want this functionality, then simply enter the 4-digit year.
SS	Yes	Second (0-59).
SSSSS	Yes	Seconds past midnight (0-86399).
TZD	Yes	Daylight savings information. The TZD value is an abbreviated time zone string with daylight savings information. It must correspond with the region specified in TZR. **Example:** PST (for US/Pacific standard time); PDT (for US/Pacific daylight time).
TZH	Yes	Time zone hour. (See TZM format element.) **Example:** 'HH:MI:SS.FFTZH:TZM'.
TZM	Yes	Time zone minute. (See TZH format element.) **Example:** 'HH:MI:SS.FFTZH:TZM'.

[a] The TO_* datetime functions are TO_CHAR, TO_DATE, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL, and TO_DSINTERVAL.

Table 2–15 (Cont.) Datetime Format Elements

Element	Specify in TO_* datetime functions?[a]	Meaning
TZR	Yes	Time zone region information. The value must be one of the time zone regions supported in the database. **Example:** US/Pacific
WW	No	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	No	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
X	Yes	Local radix character. **Example:** 'HH:MI:SSXFF'.
Y,YYY	Yes	Year with comma in this position.
YEAR SYEAR	No	Year, spelled out; "S" prefixes BC dates with "-".
YYYY SYYYY	Yes	4-digit year; "S" prefixes BC dates with "-".
YYY YY Y	Yes	Last 3, 2, or 1 digit(s) of year.

[a] The TO_* datetime functions are TO_CHAR, TO_DATE, TO_TIMESTAMP, TO_TIMESTAMP_TZ, TO_YMINTERVAL, and TO_DSINTERVAL.

Oracle returns an error if an alphanumeric character is found in the date string where punctuation character is found in the format string. For example:

```
TO_CHAR (TO_DATE('0297','MM/YY'), 'MM/YY')
```

returns an error.

Date Format Elements and Globalization Support

The functionality of some datetime format elements depends on the country and language in which you are using Oracle. For example, these datetime format elements return spelled values:

- MONTH
- MON
- DAY
- DY
- BC or AD or B.C. or A.D.
- AM or PM or A.M or P.M.

The language in which these values are returned is specified either explicitly with the initialization parameter NLS_DATE_LANGUAGE or implicitly with the initialization parameter NLS_LANGUAGE. The values returned by the YEAR and SYEAR datetime format elements are always in English.

The datetime format element D returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter NLS_TERRITORY.

> **See Also:** *Oracle9i Database Reference* and *Oracle9i Database Globalization Support Guide* for information on Globalization Support initialization parameters

ISO Standard Date Format Elements

Oracle calculates the values returned by the datetime format elements IYYY, IYY, IY, I, and IW according to the ISO standard. For information on the differences between these values and those returned by the datetime format elements YYYY, YYY, YY, Y, and WW, see the discussion of Globalization Support in *Oracle9i Database Globalization Support Guide*.

The RR Date Format Element

The RR datetime format element is similar to the YY datetime format element, but it provides additional flexibility for storing date values in other centuries. The RR datetime format element lets you store 21st century dates in the 20th century by specifying only the last two digits of the year. It will also allow you to store 20th century dates in the 21st century in the same way if necessary.

If you use the TO_DATE function with the YY datetime format element, then the date value returned always has the same first 2 digits as the current year. If you use the RR datetime format element instead, then the century of the return value varies according to the specified two-digit year and the last two digits of the current year. Table 2–16 summarizes the behavior of the RR datetime format element.

Table 2–16 The RR Date Element Format

—		If the specified two-digit year is	
		00-49	**50-99**
If the last two digits of the current year are	**00-49**	The return date has the same first 2 digits as the current date.	The first 2 digits of the return date are 1 less than the first 2 digits of the current date.
	50-99	The first 2 digits of the return date are 1 greater than the first 2 digits of the current date.	The return date has the same first 2 digits as the current date.

The following examples demonstrate the behavior of the RR datetime format element.

RR Date Format Examples

Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year"
   FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year"
   FROM DUAL;

Year
----
2017
```

Now assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year"
   FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year"
   FROM DUAL;
```

```
Year
----
2017
```

Note that the queries return the same values regardless of whether they are issued before or after the year 2000. The RR datetime format element lets you write SQL statements that will return the same values from years whose first two digits are different.

Date Format Element Suffixes

Table 2–17 lists suffixes that can be added to datetime format elements:

Table 2–17 Date Format Element Suffixes

Suffix	Meaning	Example Element	Example Value
TH	Ordinal Number	DDTH	4TH
SP	Spelled Number	DDSP	FOUR
SPTH or THSP	Spelled, ordinal number	DDSPTH	FOURTH

Restrictions:

- When you add one of these suffixes to a datetime format element, the return value is always in English.
- Date suffixes are valid only on output. You cannot use them to insert a date into the database.

Format Model Modifiers

The FM and FX modifiers, used in format models in the TO_CHAR function, control blank padding and exact format checking.

A modifier can appear in a format model more than once. In such a case, each subsequent occurrence toggles the effects of the modifier. Its effects are enabled for the portion of the model following its first occurrence, and then disabled for the portion following its second, and then reenabled for the portion following its third, and so on.

FM "Fill mode". This modifier suppresses blank padding in the return value of the TO_CHAR function:

- In a datetime format element of a TO_CHAR function, this modifier suppresses blanks in subsequent character elements (such as MONTH) and suppresses leading zeroes for subsequent number elements (such as MI) in a date format

model. Without FM, the result of a character element is always right padded with blanks to a fixed length, and leading zeroes are always returned for a number element. With FM, because there is no blank padding, the length of the return value may vary.

- In a number format element of a TO_CHAR function, this modifier suppresses blanks added to the left of the number, so that the result is left-justified in the output buffer. Without FM, the result is always right-justified in the buffer, resulting in blank-padding to the left of the number.

FX "Format exact". This modifier specifies exact matching for the character argument and date format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without FX, Oracle ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without FX, numbers in the character argument can omit leading zeroes.

  When FX is enabled, you can disable this check for leading zeroes by using the FM modifier as well.

If any portion of the character argument violates any of these conditions, then Oracle returns an error message.

Format Modifier Examples

The following statement uses a date format model to return a character expression:

```
SELECT TO_CHAR(SYSDATE, 'fmDDTH')||' of '||TO_CHAR
   (SYSDATE, 'fmMonth')||', '||TO_CHAR(SYSDATE, 'YYYY') "Ides"
    FROM DUAL;

Ides
------------------
3RD of April, 1998
```

The preceding statement also uses the FM modifier. If FM is omitted, then the month is blank-padded to nine characters:

```
SELECT TO_CHAR(SYSDATE, 'DDTH')||' of '||
   TO_CHAR(SYSDATE, 'Month')||', '||
```

```
   TO_CHAR(SYSDATE, 'YYYY') "Ides"
   FROM DUAL;

Ides
-----------------------
03RD of April    , 1998
```

The following statement places a single quotation mark in the return value by using a date format model that includes two consecutive single quotation marks:

```
SELECT TO_CHAR(SYSDATE, 'fmDay')||''''s Special' "Menu"
   FROM DUAL;

Menu
-----------------
Tuesday's Special
```

Two consecutive single quotation marks can be used for the same purpose within a character literal in a format model.

Table 2–18 shows whether the following statement meets the matching conditions for different values of char and 'fmt' using FX (the table named `table` has a column `date_column` of datatype DATE):

```
UPDATE table
  SET date_column = TO_DATE(char, 'fmt');
```

Table 2–18 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1998'	'DD-MON-YYYY'	Match
' 15! JAN % /1998'	'DD-MON-YYYY'	Error
'15/JAN/1998'	'FXDD-MON-YYYY'	Error
'15-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXDD-MON-YYYY'	Error
'01-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXFMDD-MON-YYYY'	Match

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values (*unless* you have used the FX or FXFM modifiers in the format model to control exact format checking):

- You can omit punctuation included in the format string from the date string if all the digits of the numerical format elements, including leading zeros, are specified. In other words, specify 02 and not 2 for two-digit format elements such as MM, DD, and YY.
- You can omit time fields found at the end of a format string from the date string.
- If a match fails between a datetime format element and the corresponding characters in the date string, then Oracle attempts alternative format elements, as shown in Table 2–19.

Table 2–19 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'MM'	'MON' and 'MONTH'
'MON	'MONTH'
'MONTH'	'MON'
'YY'	'YYYY'
'RR'	'RRRR'

XML Format Model

The SYS_XMLGEN function returns an instance of type XMLType containing an XML document. Oracle provides the XMLFormat object, which lets you format the output of the SYS_XMLGEN function.

Table 2–20 lists and describes the attributes of the XMLFormat object. The function that implements this type follows the table.

See Also:

- SYS_XMLGEN on page 6-163 for information on the SYS_XMLGEN function
- *Oracle9i XML API Reference - XDK and Oracle XML DB* and *Oracle9i XML Developer's Kits Guide - XDK* for more information on the implementation of the XMLFormat object and its use

Table 2–20 Attributes of the XMLFormat Object

Attribute	Datatype	Purpose
enclTag	VARCHAR2(100)	The name of the enclosing tag for the result of the SYS_XMLGEN function. If the input to the function is a column name, the default is the column name. Otherwise the default is ROW. When schemaType is set to USE_GIVEN_SCHEMA, this attribute also gives the name of the XMLSchema element.
schemaType	VARCHAR2(100)	The type of schema generation for the output document. Valid values are 'NO_SCHEMA' and 'USE_GIVEN_SCHEMA'. The default is 'NO_SCHEMA'.
schemaName	VARCHAR2(4000)	The name of the target schema Oracle uses if the value of the schemaType is 'USE_GIVEN_SCHEMA'. If you specify schemaName, then Oracle uses the enclosing tag as the element name.
processingIns	VARCHAR2(4000)	User-provided processing instructions, which are appended to the top of the function output before the element.

The function that implements the XMLFormat object follows:

```
STATIC FUNCTION createFormat(
   enclTag IN varchar2 := 'ROWSET',
   schemaType IN varchar2 := 'NO_SCHEMA',
   schemaName IN varchar2 := null,
   targetNameSpace IN varchar2 := null,
   dburlPrefix IN varchar2 := null,
   processingIns IN varchar2 := null) RETURN XMLGenFormatType,
MEMBER PROCEDURE genSchema (spec IN varchar2),
MEMBER PROCEDURE setSchemaName(schemaName IN varchar2),
MEMBER PROCEDURE setTargetNameSpace(targetNameSpace IN varchar2),
MEMBER PROCEDURE setEnclosingElementName(enclTag IN varchar2),
MEMBER PROCEDURE setDbUrlPrefix(prefix IN varchar2),
MEMBER PROCEDURE setProcessingIns(pi IN varchar2),
```

```
CONSTRUCTOR FUNCTION XMLGenFormatType (
   enclTag IN varchar2 := 'ROWSET',
   schemaType IN varchar2 := 'NO_SCHEMA',
   schemaName IN varchar2 := null,
   targetNameSpace IN varchar2 := null,
   dbUrlPrefix IN varchar2 := null,
   processingIns IN varchar2 := null) RETURN SELF AS RESULT
```

Nulls

If a column in a row has no value, then the column is said to be **null**, or to contain null. Nulls can appear in columns of any datatype that are not restricted by NOT NULL or PRIMARY KEY integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

Do not use null to represent a value of zero, because they are not equivalent. (Oracle currently treats a character value with a length of zero as null. However, this may not continue to be true in future releases, and Oracle recommends that you do not treat empty strings the same as nulls.) Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

Nulls in SQL Functions

All scalar functions (except REPLACE, NVL, and CONCAT) return null when given a null argument. You can use the NVL function to return a value when a null occurs. For example, the expression NVL(COMM,0) returns 0 if COMM is null or the value of COMM if it is not null.

Most aggregate functions ignore nulls. For example, consider a query that averages the five values 1000, null, null, null, and 2000. Such a query ignores the nulls and calculates the average to be (1000+2000)/2 = 1500.

Nulls with Comparison Conditions

To test for nulls, use only the comparison conditions IS NULL and IS NOT NULL. If you use any other condition with nulls and the result depends on the value of the null, then the result is UNKNOWN. Because null represents a lack of data, a null cannot be equal or unequal to any value or to another null. However, Oracle considers two nulls to be equal when evaluating a DECODE function.

See Also: DECODE on page 6-50 for syntax and additional information

Oracle also considers two nulls to be equal if they appear in compound keys. That is, Oracle considers identical two compound keys containing nulls if all the non-null components of the keys are equal.

Nulls in Conditions

A condition that evaluates to UNKNOWN acts almost like FALSE. For example, a SELECT statement with a condition in the WHERE clause that evaluates to UNKNOWN returns no rows. However, a condition evaluating to UNKNOWN differs from FALSE in that further operations on an UNKNOWN condition evaluation will evaluate to UNKNOWN. Thus, NOT FALSE evaluates to TRUE, but NOT UNKNOWN evaluates to UNKNOWN.

Table 2–21 shows examples of various evaluations involving nulls in conditions. If the conditions evaluating to UNKNOWN were used in a WHERE clause of a SELECT statement, then no rows would be returned for that query.

Table 2–21 Conditions Containing Nulls

If A is:	Condition	Evaluates to:
10	a IS NULL	FALSE
10	a IS NOT NULL	TRUE
NULL	a IS NULL	TRUE
NULL	a IS NOT NULL	FALSE
10	a = NULL	UNKNOWN
10	a != NULL	UNKNOWN
NULL	a = NULL	UNKNOWN
NULL	a != NULL	UNKNOWN
NULL	a = 10	UNKNOWN
NULL	a != 10	UNKNOWN

For the truth tables showing the results of logical conditions containing nulls, see Table 5–4 on page 5-8, Table 5–5 on page 5-9, and Table 5–6 on page 5-9.

Pseudocolumns

A **pseudocolumn** behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. This section describes these pseudocolumns:

- CURRVAL and NEXTVAL
- LEVEL
- ROWID
- ROWNUM
- XMLDATA

CURRVAL and NEXTVAL

A **sequence** is a schema object that can generate unique sequential values. These values are often used for primary and unique keys. You can refer to sequence values in SQL statements with these pseudocolumns:

- CURRVAL: returns the current value of a sequence
- NEXTVAL: increments the sequence and returns the next value

You must qualify CURRVAL and NEXTVAL with the name of the sequence:

```
sequence.CURRVAL
sequence.NEXTVAL
```

To refer to the current or next value of a sequence in the schema of another user, you must have been granted either SELECT object privilege on the sequence or SELECT ANY SEQUENCE system privilege, and you must qualify the sequence with the schema containing it:

```
schema.sequence.CURRVAL
schema.sequence.NEXTVAL
```

To refer to the value of a sequence on a remote database, you must qualify the sequence with a complete or partial name of a database link:

```
schema.sequence.CURRVAL@dblink
schema.sequence.NEXTVAL@dblink
```

> **See Also:** "Referring to Objects in Remote Databases" on page 2-118 for more information on referring to database links

Where to Use Sequence Values

You can use CURRVAL and NEXTVAL in:

- The SELECT list of a SELECT statement that is not contained in a subquery, materialized view, or view
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

Restrictions: You cannot use CURRVAL and NEXTVAL in the following constructs:

- A subquery in a DELETE, SELECT, or UPDATE statement
- A query of a view or of a materialized view
- A SELECT statement with the DISTINCT operator
- A SELECT statement with a GROUP BY clause or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- The condition of a CHECK constraint

Also, within a single SQL statement that uses CURRVAL or NEXTVAL, all referenced LONG columns, updated tables, and locked tables must be located on the same database.

How to Use Sequence Values

When you create a sequence, you can define its initial value and the increment between its values. The first reference to NEXTVAL returns the sequence's initial value. Subsequent references to NEXTVAL increment the sequence value by the defined increment and return the new value. Any reference to CURRVAL always returns the sequence's current value, which is the value returned by the last reference to NEXTVAL. Note that before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL.

Within a single SQL statement containing a reference to NEXTVAL, Oracle increments the sequence only once:

- For each row returned by the outer query block of a SELECT statement. Such a query block can appear in the following places:
  - A top-level SELECT statement
  - An INSERT ... SELECT statement (either single-table or multi-table). For a multi-table insert, the reference to NEXTVAL must appear in the VALUES clause, and the sequence is updated once for each row returned by the subquery, even though NEXTVAL may be referenced in multiple branches of the multi-table insert.
  - A CREATE TABLE ... AS SELECT statement
  - A CREATE MATERIALIZED VIEW ... AS SELECT statement
- For each row updated in an UPDATE statement
- For each INSERT statement containing a VALUES clause
- For row "merged" (either inserted or updated) in a MERGE statement. The reference to NEXTVAL can appear in the *merge_insert_clause* or the *merge_update_clause*.

If any of these locations contains more than one reference to NEXTVAL, then Oracle increments the sequence once and returns the same value for all occurrences of NEXTVAL.

If any of these locations contains references to both CURRVAL and NEXTVAL, then Oracle increments the sequence and returns the same value for both CURRVAL and NEXTVAL.

A sequence can be accessed by many users concurrently with no waiting or locking.

> **See Also:** CREATE SEQUENCE on page 14-87 for information on sequences

Finding the current value of a sequence: Example This example selects the next value of the employee sequence in the sample schema hr:

```
SELECT employees_seq.nextval
    FROM DUAL;
```

Inserting sequence values into a table: Example This example increments the employee sequence and uses its value for a new employee inserted into the sample table hr.employees:

```
INSERT INTO employees
```

```
   VALUES (employees_seq.nextval, 'John', 'Doe', 'jdoe',
   '555-1212', TO_DATE(SYSDATE), 'PU_CLERK', 2500, null, null,
   30);
```

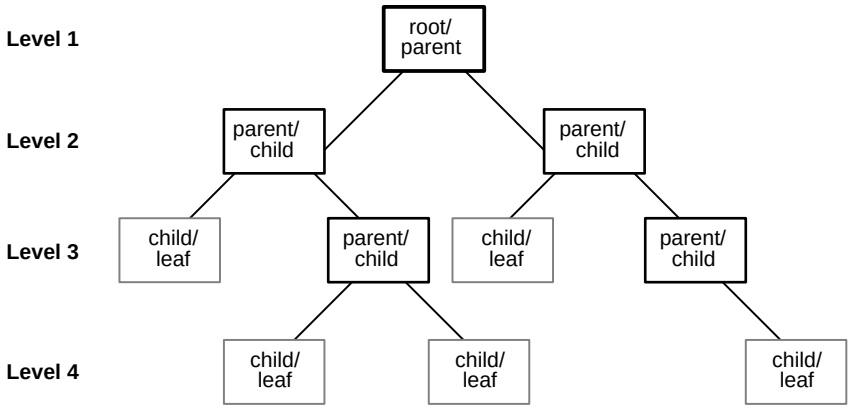
Reusing the current value of a sequence: Example This example adds a new order with the next order number to the master order table. It then adds suborders with this number to the detail order table:

```
INSERT INTO orders (order_id, order_date, customer_id)
   VALUES (orders_seq.nextval, TO_DATE(SYSDATE), 106);

INSERT INTO order_items (order_id, line_item_id, product_id)
   VALUES (orders_seq.currval, 1, 2359);

INSERT INTO order_items (order_id, line_item_id, product_id)
   VALUES (orders_seq.currval, 2, 3290);

INSERT INTO order_items (order_id, line_item_id, product_id)
   VALUES (orders_seq.currval, 3, 2381);
```

LEVEL

For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on. A **root row** is the highest row within an inverted tree. A **child row** is any nonroot row. A **parent row** is any row that has children. A **leaf row** is any row without children. Figure 2–1 shows the nodes of an inverted tree with their LEVEL values.

Figure 2–1 Hierarchical Tree



To define a hierarchical relationship in a query, you must use the START WITH and CONNECT BY clauses.

Restriction on LEVEL in WHERE clauses: In a [NOT] IN condition in a WHERE clause, if the right-hand side of the condition is a subquery, you cannot use LEVEL on the left-hand side of the condition. However, you can specify LEVEL in a subquery of the FROM clause to achieve the same result. For example, the following statement is not valid:

```
SELECT employee_id, last_name FROM employees
   WHERE (employee_id, LEVEL)
      IN (SELECT employee_id, 2 FROM employees)
   START WITH employee_id = 2
   CONNECT BY PRIOR employee_id = manager_id;
```

But the following statement is valid because it encapsulates the query containing the LEVEL information in the FROM clause:

```
SELECT v.employee_id, v.last_name, v.lev
   FROM
      (SELECT employee_id, last_name, LEVEL lev
      FROM employees v
      START WITH employee_id = 100
      CONNECT BY PRIOR employee_id = manager_id) v
   WHERE (v.employee_id, v.lev) IN
      (SELECT employee_id, 2 FROM employees);
```

> **See Also:** "Hierarchical Queries" on page 8-3 for information on hierarchical queries in general

ROWID

For each row in the database, the ROWID pseudocolumn returns a row's address. Oracle9*i* rowid values contain information necessary to locate a row:

- The data object number of the object
- Which data block in the datafile
- Which row in the data block (first row is 0)
- Which datafile (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.

Values of the ROWID pseudocolumn have the datatype ROWID or UROWID.

> **See Also:** "ROWID Datatype" on page 2-33 and "UROWID Datatype" on page 2-35

Rowid values have several important uses:

- They are the fastest way to access a single row.
- They can show you how a table's rows are stored.
- They are unique identifiers for rows in a table.

You should not use ROWID as a table's primary key. If you delete and reinsert a row with the Import and Export utilities, for example, then its rowid may change. If you delete a row, then Oracle may reassign its rowid to a new row inserted later.

Although you can use the ROWID pseudocolumn in the SELECT and WHERE clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the ROWID pseudocolumn.

Example This statement selects the address of all rows that contain data for employees in department 20:

```
SELECT ROWID, last_name
   FROM employees
   WHERE department_id = 20;
```

ROWNUM

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

You can use ROWNUM to limit the number of rows returned by a query, as in this example:

```
SELECT * FROM employees WHERE ROWNUM < 10;
```

If an ORDER BY clause follows ROWNUM in the same query, then the rows will be reordered by the ORDER BY clause. The results can vary depending on the way the rows are accessed. For example, if the ORDER BY clause causes Oracle to use an index to access the data, then Oracle may retrieve the rows in a different order than without the index. Therefore, the following statement will not have the same effect as the preceding example:

```
SELECT * FROM employees WHERE ROWNUM < 11 ORDER BY last_name;
```

If you embed the ORDER BY clause in a subquery and place the ROWNUM condition in the top-level query, then you can force the ROWNUM condition to be applied after the ordering of the rows. For example, the following query returns the 10 smallest employee numbers. This is sometimes referred to as a "top-N query":

```
SELECT * FROM
   (SELECT * FROM employees ORDER BY employee_id)
   WHERE ROWNUM < 11;
```

In the preceding example, the ROWNUM values are those of the top-level SELECT statement, so they are generated after the rows have already been ordered by employee_id in the subquery.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about top-N queries

Conditions testing for ROWNUM values greater than a positive integer are always false. For example, this query returns no rows:

```
SELECT * FROM employees
    WHERE ROWNUM > 1;
```

The first row fetched is assigned a ROWNUM of 1 and makes the condition false. The second row to be fetched is now the first row and is also assigned a ROWNUM of 1

and makes the condition false. All rows subsequently fail to satisfy the condition, so no rows are returned.

You can also use ROWNUM to assign unique values to each row of a table, as in this example:

```
UPDATE my_table
    SET column1 = ROWNUM;
```

Note: Using ROWNUM in a query can affect view optimization. For more information, see *Oracle9i Database Concepts*.

XMLDATA

Oracle stores XMLType data either in LOB or object-relational columns, based on XMLSchema information and how you specify the storage clause. The XMLDATA pseudocolumn lets you access the underlying LOB or object relational column to specify additional storage clause parameters, constraints, indexes, and so forth.

Example The following statements illustrate the use of this pseudocolumn. Suppose you create a simple table of XMLType:

```
CREATE TABLE xml_lob_tab of XMLTYPE;
```

The default storage is in a CLOB column. To change the storage characteristics of the underlying LOB column, you can use the following statement:

```
ALTER TABLE xml_lob_tab MODIFY LOB (XMLDATA)
   (STORAGE (BUFFER_POOL DEFAULT) CACHE);
```

Now suppose you have created an XMLSchema-based table like the xwarehouses table created in "Using XML in SQL Statements" on page D-11. You could then use the XMLDATA column to set the properties of the underlying columns, as shown in the following statement:

```
ALTER TABLE xwarehouses ADD (UNIQUE(XMLDATA."WarehouseId"));
```

Comments

You can associate comments with SQL statements and schema objects.

Comments Within SQL Statements

Comments within SQL statements do not affect the statement execution, but they may make your application easier for you to read and maintain. You may want to include a comment in a statement that describes the statement's purpose within your application.

A comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement using either of these means:

- Begin the comment with a slash and an asterisk (/*). Proceed with the text of the comment. This text can span multiple lines. End the comment with an asterisk and a slash (*/). The opening and terminating characters need not be separated from the text by a space or a line break.
- Begin the comment with -- (two hyphens). Proceed with the text of the comment. This text cannot extend to a new line. End the comment with a line break.

A SQL statement can contain multiple comments of both styles. The text of a comment can contain any printable characters in your database character set.

Example These statements contain many comments:

```
SELECT last_name, salary + NVL(commission_pct, 0),
   job_id, e.department_id
/* Select all employees whose compensation is
greater than that of Pataballa.*/
  FROM employees e, departments d
       /*The DEPARTMENTS table is used to get the department name.*/
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct,0) >   /* Subquery:       */
   (SELECT salary + NVL(commission_pct,0)
                 /* total compensation is salar + commission_pct */
      FROM employees
      WHERE last_name = 'Pataballa');

SELECT last_name,                     -- select the name
    salary + NVL(commission_pct, 0),-- total compensation
    job_id,                          -- job
    e.department_id                  -- and department
  FROM employees e,                  -- of all employees
       departments d
  WHERE e.department_id = d.department_id
    AND salary + NVL(commission_pct, 0) >  -- whose compensation
```

```
                                          -- is greater than
      (SELECT salary + NVL(commission_pct,0)  -- the compensation
    FROM employees
    WHERE last_name = 'Pataballa')        -- of Pataballa.
;
```

Comments on Schema Objects

You can associate a comment with a table, view, materialized view, or column using the COMMENT command. Comments associated with schema objects are stored in the data dictionary.

See Also: COMMENT on page 12-69 for a description of comments

Hints

You can use comments in a SQL statement to pass instructions, or **hints**, to the Oracle optimizer. The optimizer uses these hints as suggestions for choosing an execution plan for the statement.

A statement block can have only one comment containing hints, and that comment must follow the SELECT, UPDATE, INSERT, or DELETE keyword. The following syntax shows hints contained in both styles of comments that Oracle supports within a statement block.

```
{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */
```

or

```
{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...
```

where:

- DELETE, INSERT, SELECT, or UPDATE is a DELETE, INSERT, SELECT, or UPDATE keyword that begins a statement block. Comments containing hints can appear only after these keywords.
- + is a plus sign that causes Oracle to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter (no space is permitted).
- *hint* is one of the hints discussed in this section. The space between the plus sign and the hint is optional. If the comment contains multiple hints, then separate the hints by at least one space.
- *text* is other commenting text that can be interspersed with the hints.

Table 2–22 lists the hints by functional category. An alphabetical listing of the hints, including the syntax and a brief description of each hint, follow the table.

Note: Oracle treats misspelled hints as regular comments and does not return an error.

See Also: *Oracle9i Database Performance Tuning Guide and Reference* and *Oracle9i Database Concepts* for more information on hints

Table 2–22 Hints by Functional Category

Category	Hint
Optimization Goals and Approaches	ALL_ROWS and FIRST_ROWS
	CHOOSE
	RULE
Access Method Hints	AND_EQUAL
	CLUSTER
	FULL
	HASH
	INDEX and NO_INDEX
	INDEX_ASC and INDEX_DESC
	INDEX_COMBINE
	INDEX_FFS
	ROWID
Join Order Hints	ORDERED
	STAR
Join Operation Hints	DRIVING_SITE
	HASH_SJ, MERGE_SJ, and NL_SJ
	LEADING
	USE_HASH and USE_MERGE
	USE_NL

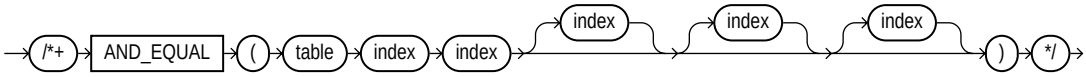
Table 2–22 (Cont.) Hints by Functional Category

Category	Hint
Parallel Execution Hints	PARALLEL and NOPARALLEL
	PARALLEL_INDEX
	PQ_DISTRIBUTE
	NOPARALLEL_INDEX
Query Transformation Hints	EXPAND_GSET_TO_UNION
	FACT and NOFACT
	MERGE
	NO_EXPAND
	NO_MERGE
	REWRITE and NOREWRITE
	STAR_TRANSFORMATION
	USE_CONCAT
Other Hints	APPEND and NOAPPEND
	CACHE and NOCACHE
	CURSOR_SHARING_EXACT
	DYNAMIC_SAMPLING
	NESTED_TABLE_GET_REFS
	UNNEST and NO_UNNEST
	ORDERED_PREDICATES
	PUSH_PRED and NO_PUSH_PRED
	PUSH_SUBQ

all_rows_hint::=

The ALL_ROWS hint explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).

and_equal_hint::=



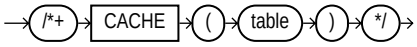
The AND_EQUAL hint explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes.

append_hint::=

The APPEND hint lets you enable direct-path INSERT if your database is running in serial mode. (Your database is in serial mode if you are not using Enterprise Edition. Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode).

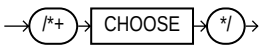
In direct-path INSERT, data is appended to the end of the table, rather than using existing space currently allocated to the table. As a result, direct-path INSERT can be considerably faster than conventional INSERT.

cache_hint::=



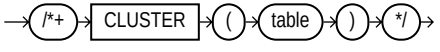
The CACHE hint specifies that the blocks retrieved for the table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This option is useful for small lookup tables.

choose_hint::=



The CHOOSE hint causes the optimizer to choose between the rule-based and cost-based approaches for a SQL statement. The optimizer bases its selection on the presence of statistics for the tables accessed by the statement. If the data dictionary has statistics for at least one of these tables, then the optimizer uses the cost-based approach and optimizes with the goal of best throughput. If the data dictionary does not have statistics for these tables, then it uses the rule-based approach.

cluster_hint::=



The CLUSTER hint explicitly chooses a cluster scan to access the specified table. It applies only to clustered objects.

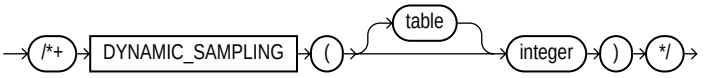
cursor_sharing_exact_hint::=

Oracle can replace literals in SQL statements with bind variables, if it is safe to do so. This is controlled with the CURSOR_SHARING startup parameter. The CURSOR_SHARING_EXACT hint causes this behavior to be switched off. In other words, Oracle executes the SQL statement without any attempt to replace literals by bind variables.

driving_site_hint::=

The DRIVING_SITE hint forces query execution to be done at a different site than that selected by Oracle. This hint can be used with either rule-based or cost-based optimization.

dynamic_sampling_hint::=

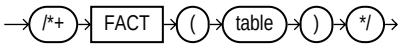


The DYNAMIC_SAMPLING hint lets you control dynamic sampling to improve server performance by determining more accurate selectivity and cardinality estimates. You can set the value of DYNAMIC_SAMPLING to a value from 0 to 10. The higher the level, the more effort the compiler puts into dynamic sampling and the more broadly it is applied. Sampling defaults to cursor level unless you specify a table.

expand_gset_to_union_hint::=

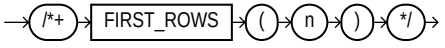
The EXPAND_GSET_TO_UNION hint is used for queries containing grouping sets (such as queries with GROUP BY GROUPING SET or GROUP BY ROLLUP). The hint forces a query to be transformed into a corresponding query with UNION ALL of individual groupings.

fact_hint::=



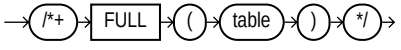
The FACT hint is used in the context of the star transformation to indicate to the transformation that the hinted table should be considered as a fact table.

first_rows_hint::=



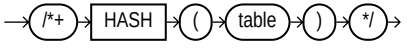
The hints FIRST_ROWS(*n*) (where *n* is any positive integer) or FIRST_ROWS instruct Oracle to optimize an individual SQL statement for fast response. FIRST_ROWS(n) affords greater precision, because it instructs Oracle to choose the plan that returns the first *n* rows most efficiently. The FIRST_ROWS hint, which optimizes for the best plan to return the first single row, is retained for backward compatibility and plan stability.

full_hint::=



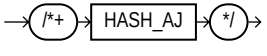
The FULL hint explicitly chooses a full table scan for the specified table.

hash_hint::=



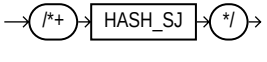
The HASH hint explicitly chooses a hash scan to access the specified table. It applies only to tables stored in a cluster.

hash_aj_hint::=



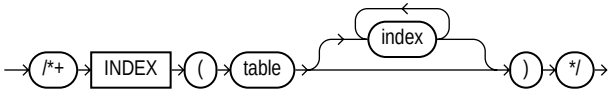
For a specific query, place the HASH_SJ, MERGE_SJ, or NL_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join, MERGE_SJ uses a sort merge semi-join, and NL_SJ uses a nested loop semi-join.

hash_sj_hint::=



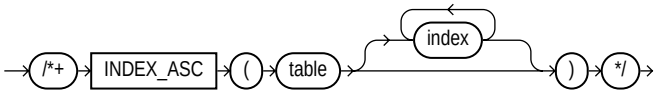
For a specific query, place the HASH_SJ, MERGE_SJ, or NL_SJ hint into the EXISTS subquery. HASH_SJ uses a hash semi-join, MERGE_SJ uses a sort merge semi-join, and NL_SJ uses a nested loop semi-join.

index_hint::=



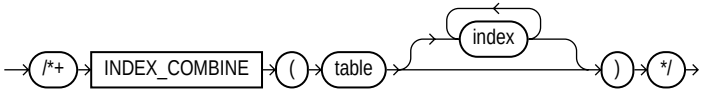
The INDEX hint explicitly chooses an index scan for the specified table. You can use the INDEX hint for domain, B-tree, bitmap, and bitmap join indexes. However, Oracle recommends using INDEX_COMBINE rather than INDEX for bitmap indexes, because it is a more versatile hint.

index_asc_hint::=



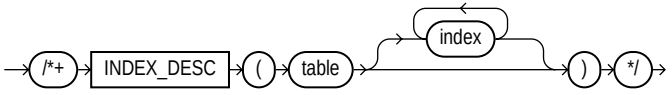
The INDEX_ASC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in ascending order of their indexed values.

index_combine_hint::=



The INDEX_COMBINE hint explicitly chooses a bitmap access path for the table. If no indexes are given as arguments for the INDEX_COMBINE hint, then the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate for the table. If certain indexes are given as arguments, then the optimizer tries to use some Boolean combination of those particular bitmap indexes.

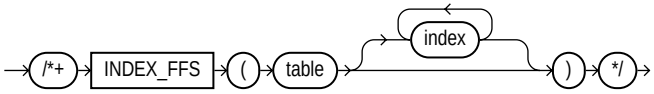
index_desc_hint::=



The INDEX_DESC hint explicitly chooses an index scan for the specified table. If the statement uses an index range scan, then Oracle scans the index entries in

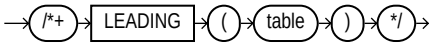
descending order of their indexed values. In a partitioned index, the results are in descending order within each partition.

index_ffs_hint::=



The INDEX_FFS hint causes a fast full index scan to be performed rather than a full table scan.

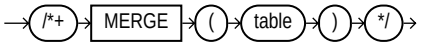
leading_hint::=



The LEADING hint causes Oracle to use the specified table as the first table in the join order.

If you specify two or more LEADING hints on different tables, then all of them are ignored. If you specify the ORDERED hint, then it overrides all LEADING hints.

merge_hint::=



The MERGE hint lets you merge a view for each query.

If a view's query contains a GROUP BY clause or DISTINCT operator in the SELECT list, then the optimizer can merge the view's query into the accessing statement only if complex view merging is enabled. Complex merging can also be used to merge an IN subquery into the accessing statement if the subquery is uncorrelated.

Complex merging is not cost-based; that is, the accessing query block must include the MERGE hint. Without this hint, the optimizer uses another approach.

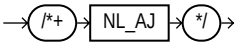
merge_aj_hint::=

See HASH_AJ hint.

merge_sj_hint::=

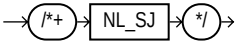
See HASH_SJ hint.

nl_aj_hint::=



See HASH_AJ hint.

nl_sj_hint::=

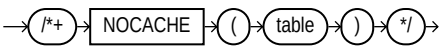


See HASH_SJ hint.

noappend_hint::=

The NOAPPEND hint enables conventional INSERT by disabling parallel mode for the duration of the INSERT statement. (Conventional INSERT is the default in serial mode, and direct-path INSERT is the default in parallel mode).

nocache_hint::=



The NOCACHE hint specifies that the blocks retrieved for the table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of blocks in the buffer cache.

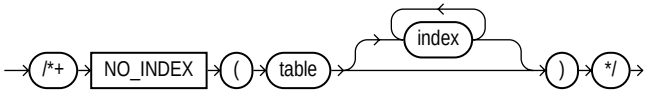
no_expand_hint::=

The NO_EXPAND hint prevents the cost-based optimizer from considering OR-expansion for queries having OR conditions or IN-lists in the WHERE clause. Usually, the optimizer considers using OR expansion and uses this method if it decides that the cost is lower than not using it.

no_fact_hint::=

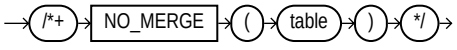
The NO_FACT hint is used in the context of the star transformation to indicate to the transformation that the hinted table should not be considered as a fact table.

no_index_hint::=



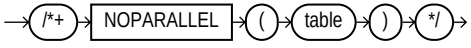
The NO_INDEX hint explicitly disallows a set of indexes for the specified table.

no_merge_hint::=



The NO_MERGE hint causes Oracle not to merge mergeable views.

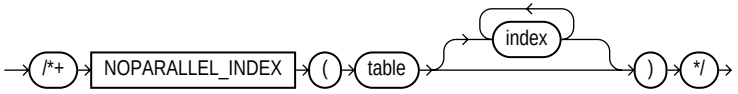
noparallel_hint::=



The NOPARALLEL hint overrides a PARALLEL specification in the table clause. In general, hints take precedence over table clauses.

Restriction: You cannot parallelize a query involving a nested table.

noparallel_index_hint::=

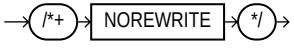


The NOPARALLEL_INDEX hint overrides a PARALLEL attribute setting on an index to avoid a parallel index scan operation.

no_push_pred_hint::=

The NO_PUSH_PRED hint prevents pushing of a join predicate into the view.

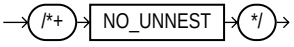
norewrite_hint::=



The NOREWRITE hint disables query rewrite for the query block, overriding the setting of the parameter QUERY_REWRITE_ENABLED. Use the NOREWRITE hint on any query block of a request.

Note: The NOREWRITE hint disables the use of function-based indexes.

no_unnest_hint::=



Use of the NO_UNNEST hint turns off unnesting for specific subquery blocks.

ordered_hint::=

The ORDERED hint causes Oracle to join tables in the order in which they appear in the FROM clause.

If you omit the ORDERED hint from a SQL statement performing a join, then the optimizer chooses the order in which to join the tables. You might want to use the ORDERED hint to specify a join order if you know something about the number of rows selected from each table that the optimizer does not. Such information lets you choose an inner and outer table better than the optimizer could.

ordered_predicates_hint::=

The ORDERED_PREDICATES hint forces the optimizer to preserve the order of predicate evaluation, except for predicates used as index keys. Use this hint in the WHERE clause of SELECT statements.

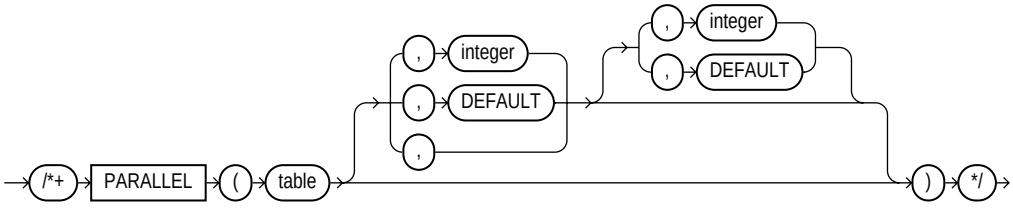
If you do not use the ORDERED_PREDICATES hint, then Oracle evaluates all predicates in the following order:

1. Predicates without user-defined functions, type methods, or subqueries are evaluated first, in the order specified in the WHERE clause.
2. Predicates with user-defined functions and type methods that have user-computed costs are evaluated next, in increasing order of their cost.

3. Predicates with user-defined functions and type methods without user-computed costs are evaluated next, in the order specified in the WHERE clause.
4. Predicates not specified in the WHERE clause (for example, predicates transitively generated by the optimizer) are evaluated next.
5. Predicates with subqueries are evaluated last, in the order specified in the WHERE clause.

> **Note:** Remember, you cannot use the ORDERED_PREDICATES hint to preserve the order of predicate evaluation on index keys.

parallel_hint::=



The PARALLEL hint lets you specify the desired number of concurrent servers that can be used for a parallel operation. The hint applies to the SELECT, INSERT, UPDATE, and DELETE portions of a statement, as well as to the table scan portion.

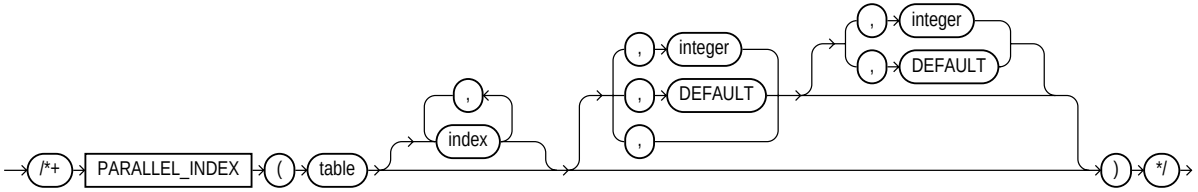
> **Note:** The number of servers that can be used is twice the value in the PARALLEL hint, if sorting or grouping operations also take place.

If any parallel restrictions are violated, then the hint is ignored.

> **Note:** Oracle ignores parallel hints on a temporary table.

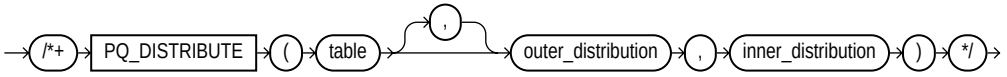
> **See Also:** CREATE TABLE on page 15-7 and *Oracle9i Database Concepts*

parallel_index_hint::=



The PARALLEL_INDEX hint specifies the desired number of concurrent servers that can be used to parallelize index range scans for partitioned indexes.

pq_distribute_hint::=

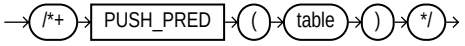


The PQ_DISTRIBUTE hint improves the performance of parallel join operations. Do this by specifying how rows of joined tables should be distributed among producer and consumer query servers. Using this hint overrides decisions the optimizer would normally make.

Use the EXPLAIN PLAN statement to identify the distribution chosen by the optimizer. The optimizer ignores the distribution hint, if both tables are serial.

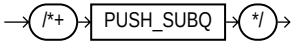
> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for the permitted combinations of distributions for the outer and inner join tables

push_pred_hint::=



The PUSH_PRED hint forces pushing of a join predicate into the view.

push_subq_hint::=

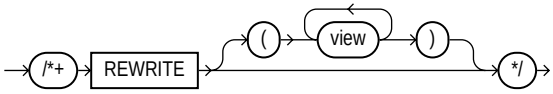


The PUSH_SUBQ hint causes non-merged subqueries to be evaluated at the earliest possible step in the execution plan. Generally, subqueries that are not merged are executed as the last step in the execution plan. If the subquery is relatively

inexpensive and reduces the number of rows significantly, then it improves performance to evaluate the subquery earlier.

This hint has no effect if the subquery is applied to a remote table or one that is joined using a merge join.

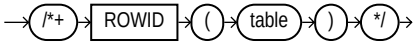
rewrite_hint::=



The REWRITE hint forces the cost-based optimizer to rewrite a query in terms of materialized views, when possible, without cost consideration. Use the REWRITE hint with or without a view list. If you use REWRITE with a view list and the list contains an eligible materialized view, then Oracle uses that view regardless of its cost.

Oracle does not consider views outside of the list. If you do not specify a view list, then Oracle searches for an eligible materialized view and always uses it regardless of its cost.

rowid_hint::=



The ROWID hint explicitly chooses a table scan by rowid for the specified table.

rule_hint::=

The RULE hint explicitly chooses rule-based optimization for a statement block. It also makes the optimizer ignore other hints specified for the statement block.

star_hint::=

The STAR hint forces a star query plan to be used, if possible. A star plan has the largest table in the query last in the join order and joins it with a nested loops join on a concatenated index. The STAR hint applies when there are at least three tables, the large table's concatenated index has at least three columns, and there are no

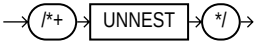
conflicting access or join method hints. The optimizer also considers different permutations of the small tables.

star_transformation_hint::=

The STAR_TRANSFORMATION hint makes the optimizer use the best plan in which the transformation has been used. Without the hint, the optimizer could make a cost-based decision to use the best plan generated without the transformation, instead of the best plan for the transformed query.

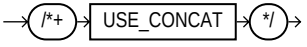
Even if the hint is given, there is no guarantee that the transformation will take place. The optimizer only generates the subqueries if it seems reasonable to do so. If no subqueries are generated, then there is no transformed query, and the best plan for the untransformed query is used, regardless of the hint.

unnest_hint::=



The UNNEST hint tells Oracle to check the subquery block for validity only. If the subquery block is valid, then subquery unnesting is enabled without Oracle's checking the heuristics.

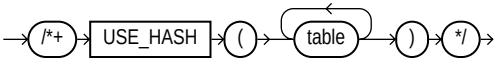
use_concat_hint::=



The USE_CONCAT hint forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator. Generally, this transformation occurs only if the cost of the query using the concatenations is cheaper than the cost without them.

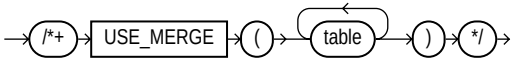
The USE_CONCAT hint turns off IN-list processing and OR-expands all disjunctions, including IN-lists.

use_hash_hint::=



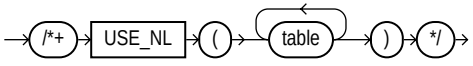
The USE_HASH hint causes Oracle to join each specified table with another row source, using a hash join.

use_merge_hint::=



The USE_MERGE hint causes Oracle to join each specified table with another row source, using a sort-merge join.

use_nl_hint::=



The USE_NL hint causes Oracle to join each specified table to another row source with a nested loops join, using the specified table as the inner table.

Database Objects

Oracle recognizes objects that are associated with a particular schema and objects that are not associated with a particular schema, as described in the sections that follow.

Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

Clusters
Constraints
Database links
Database triggers
Dimensions
External procedure libraries
Index-organized tables
Indexes
Indextypes
Java classes, Java resources, Java sources
Materialized views
Materialized view logs
Object tables
Object types

Object views
Operators
Packages
Sequences
Stored functions, stored procedures
Synonyms
Tables
Views

Nonschema Objects

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

Contexts
Directories
Parameter files (PFILEs) and server parameter files (SPFILEs)
Profiles
Roles
Rollback segments
Tablespaces
Users

In this reference, each type of object is briefly defined in Chapter 9 through Chapter 18, in the section describing the statement that creates the database object. These statements begin with the keyword CREATE. For example, for the definition of a cluster, see CREATE CLUSTER on page 13-2.

> **See Also:** *Oracle9i Database Concepts* for an overview of database objects

You must provide names for most types of database objects when you create them. These names must follow the rules listed in the following sections.

Parts of Schema Objects

Some schema objects are made up of parts that you can or must name, such as:

- Columns in a table or view
- Index and table partitions and subpartitions
- Integrity constraints on a table

- Packaged procedures, packaged stored functions, and other objects stored within a package

Partitioned Tables and Indexes

Tables and indexes can be partitioned. When partitioned, these schema objects consist of a number of parts called **partitions**, all of which have the same logical attributes. For example, all partitions in a table share the same column and constraint definitions, and all partitions in an index share the same index columns.

When you partition a table or index using the range method, you specify a maximum value for the partitioning key column(s) for each partition. When you partition a table or index using the list method, you specify actual values for the partitioning key column(s) for each partition. When you partition a table or index using the hash method, you instruct Oracle to distribute the rows of the table into partitions based on a system-defined hash function on the partitioning key column(s). When you partition a table or index using the composite-partitioning method, you specify ranges for the partitions, and Oracle distributes the rows in each partition into one or more hash subpartitions based on a hash function. Each subpartition of a table or index partitioned using the composite method has the same logical attributes.

Partition-Extended and Subpartition-Extended Names

Partition-extended and subpartition-extended names let you perform some partition-level and subpartition-level operations, such as deleting all rows from a partition or subpartition, on only one partition or subpartition. Without extended names, such operations would require that you specify a predicate (`WHERE` clause). For range- and list-partitioned tables, trying to phrase a partition-level operation with a predicate can be cumbersome, especially when the range partitioning key uses more than one column. For hash partitions and subpartitions, using a predicate is more difficult still, because these partitions and subpartitions are based on a system-defined hash function.

Partition-extended names let you use partitions as if they were tables. An advantage of this method, which is most useful for range-partitioned tables, is that you can build partition-level access control mechanisms by granting (or revoking) privileges on these views to (or from) other users or roles.To use a partition as a table, create a view by selecting data from a single partition, and then use the view as a table.

You can specify partition-extended or subpartition-extended table names for the following DML statements:

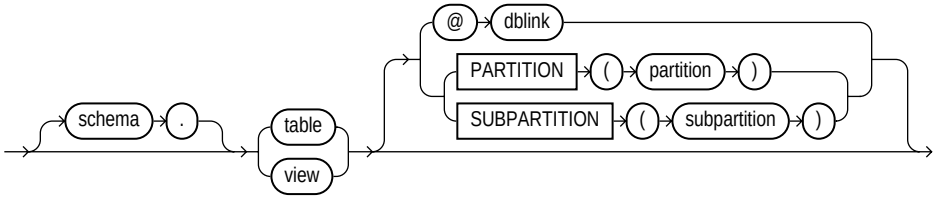
- `DELETE`

- INSERT
- LOCK TABLE
- SELECT
- UPDATE

> **Note:** For application portability and ANSI syntax compliance, Oracle strongly recommends that you use views to insulate applications from this Oracle proprietary extension.

Syntax The basic syntax for using partition-extended and subpartition-extended table names is:

partition_extended_name::=



Restrictions Currently, the use of partition-extended and subpartition-extended table names has the following restrictions:

- No remote tables: A partition-extended or subpartition-extended table name cannot contain a database link (dblink) or a synonym that translates to a table with a dblink. To use remote partitions and subpartitions, create a view at the remote site that uses the extended table name syntax and then refer to the remote view.
- No synonyms: A partition or subpartition extension must be specified with a base table. You cannot use synonyms, views, or any other objects.

Example In the following statement, `sales` is a partitioned table with partition `sales_q1_2000`. You can create a view of the single partition `sales_q1_2000`, and then use it as if it were a table. This example deletes rows from the partition.

```
CREATE VIEW Q1_2000_sales AS
   SELECT * FROM sales PARTITION (SALES_Q1_2000);
```

```
DELETE FROM Q1_2000_sales WHERE amount_sold < 0;
```

Schema Object Names and Qualifiers

This section provides:

- Rules for naming schema objects and schema object location qualifiers
- Guidelines for naming schema objects and qualifiers

Schema Object Naming Rules

Every database object has a name. In a SQL statement, you represent the name of an object with a **quoted identifier** or a **nonquoted identifier**.

- A quoted identifier begins and ends with double quotation marks ("). If you name a schema object using a quoted identifier, then you must use the double quotation marks whenever you refer to that object.
- A nonquoted identifier is not surrounded by any punctuation.

You can use either quoted or nonquoted identifiers to name any database object, with one exception: database links must be named with nonquoted identifiers. In addition, Oracle Corporation strongly recommends that you not use quotation marks to make usernames and passwords case sensitive.

> **See Also:** CREATE USER on page 16-32 for additional rules for naming users and passwords

The following list of rules applies to both quoted and nonquoted identifiers unless otherwise indicated:

1. Names must be from 1 to 30 bytes long with these exceptions:
   - Names of databases are limited to 8 bytes.
   - Names of database links can be as long as 128 bytes.
2. Nonquoted identifiers cannot be Oracle reserved words. Quoted identifiers can be reserved words, although this is not recommended.

   Depending on the Oracle product you plan to use to access a database object, names might be further restricted by other product-specific reserved words.

Note: The reserved word `ROWID` is an exception to this rule. You cannot use the uppercase word `ROWID` as a name, even in double quotation marks. However, you can use the word with one or more lower case letters (for example, "`Rowid`" or "`rowid`").

See Also:

- Appendix C, "Oracle Reserved Words" for a listing of all Oracle reserved words
- The manual for the specific product, such as *PL/SQL User's Guide and Reference*, for a list of the product's reserved words

3. The Oracle SQL language contains other words that have special meanings. These words include datatypes, function names, the dummy system table `DUAL`, and keywords (the uppercase words in SQL statements, such as `DIMENSION`, `SEGMENT`, `ALLOCATE`, `DISABLE`, and so forth). These words are not reserved. However, Oracle uses them internally in specific ways. Therefore, if you use these words as names for objects and object parts, then your SQL statements may be more difficult to read and may lead to unpredictable results.

   In particular, do not use words beginning with "`SYS_`" as schema object names, and do not use the names of SQL built-in functions for the names of schema objects or user-defined functions.

   See Also:

   - "Datatypes" on page 2-2 and "SQL Functions" on page 6-2
   - "Selecting from the DUAL Table" on page 8-15

4. You should use ASCII characters in database names, global database names, and database link names, because ASCII characters provide optimal compatibility across different platforms and operating systems.

   Note: Oracle Corporation recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform. Please refer to *Oracle9i Database Administrator's Guide* for more information about this recommendation.

   ---

5. Nonquoted identifiers must begin with an alphabetic character from your database character set. Quoted identifiers can begin with any character.

6. Nonquoted identifiers can contain only alphanumeric characters from your database character set and the underscore (_), dollar sign ($), and pound sign (#). Database links can also contain periods (.) and "at" signs (@). Oracle Corporation strongly discourages you from using $ and #.

   Quoted identifiers can contain any characters and punctuations marks as well as spaces. However, neither quoted nor nonquoted identifiers can contain double quotation marks.

7. Within a namespace, no two objects can have the same name.

   The following schema objects share one namespace:

   - Tables
   - Views
   - Sequences
   - Private synonyms
   - Stand-alone procedures
   - Stand-alone stored functions
   - Packages
   - Materialized views
   - User-defined types

   Each of the following schema objects has its own namespace:

   - Indexes
   - Constraints
   - Clusters
   - Database triggers
   - Private database links
   - Dimensions

   Because tables and views are in the same namespace, a table and a view in the same schema cannot have the same name. However, tables and indexes are in different namespaces. Therefore, a table and an index in the same schema can have the same name.

Each schema in the database has its own namespaces for the objects it contains. This means, for example, that two tables in different schemas are in different namespaces and can have the same name.

Each of the following nonschema objects also has its own namespace:

- User roles
- Public synonyms
- Public database links
- Tablespaces
- Rollback segments
- Profiles
- Parameter files (PFILEs) and server parameter files (SPFILEs)

Because the objects in these namespaces are not contained in schemas, these namespaces span the entire database.

8. Nonquoted identifiers are not case sensitive. Oracle interprets them as uppercase. Quoted identifiers are case sensitive.

By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

```
employees
"employees"
"Employees"
"EMPLOYEES"
```

Note that Oracle interprets the following names the same, so they cannot be used for different objects in the same namespace:

```
employees
EMPLOYEES
"EMPLOYEES"
```

9. If you name a user or a password with a quoted identifier, then the name cannot contain lowercase letters.

10. Columns in the same table or view cannot have the same name. However, columns in different tables or views can have the same name.

11. Procedures or functions contained in the same package can have the same name, if their arguments are not of the same number and datatypes. Creating

multiple procedures or functions with the same name in the same package with different arguments is called **overloading** the procedure or function.

Schema Object Naming Examples

The following examples are valid schema object names:

```
last_name
horse
hr.hire_date
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

All of these examples adhere to the rules listed in "Schema Object Naming Rules" on page 2-111. The following example is not valid, because it exceeds 30 characters:

```
a_very_very_long_and_valid_name
```

Although column aliases, table aliases, usernames, and passwords are not objects or parts of objects, they must also follow these naming rules unless otherwise specified in the rules themselves.

Schema Object Naming Guidelines

Here are several helpful guidelines for naming objects and their parts:

- Use full, descriptive, pronounceable names (or well-known abbreviations).
- Use consistent naming rules.
- Use the same name to describe the same entity or attribute across tables.

When naming objects, balance the objective of keeping names short and easy to use with the objective of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in the database may be used by many people over a period of time. Your counterpart ten years from now may have difficulty understanding a table column with a name like `pmdd` instead of `payment_due_date`.

Using consistent naming rules helps users understand the part that each table plays in your application. One such rule might be to begin the names of all tables belonging to the `FINANCE` application with `fin_`.

Use the same names to describe the same things across tables. For example, the department number columns of the sample `employees` and `departments` tables are both named `deptno`.

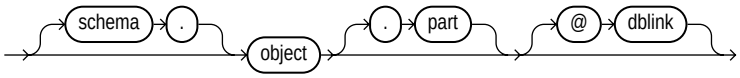
Syntax for Schema Objects and Parts in SQL Statements

This section tells you how to refer to schema objects and their parts in the context of a SQL statement. This section shows you:

- The general syntax for referring to an object
- How Oracle resolves a reference to an object
- How to refer to objects in schemas other than your own
- How to refer to objects in remote databases

The following diagram shows the general syntax for referring to an object or a part:

object_part::=



where:

- *object* is the name of the object.
- *schema* is the schema containing the object. The schema qualifier lets you refer to an object in a schema other than your own. You must be granted privileges to refer to objects in other schemas. If you omit *schema*, then Oracle assumes that you are referring to an object in your own schema.

  Only schema objects can be qualified with *schema*. Schema objects are shown with list item 7 on page 2-113. Nonschema objects, also shown with list item 7 on page 2-113, cannot be qualified with *schema* because they are not schema objects. (An exception is public synonyms, which can optionally be qualified with "PUBLIC". The quotation marks are required.)
- *part* is a part of the object. This identifier lets you refer to a part of a schema object, such as a column or a partition of a table. Not all types of objects have parts.
- *dblink* applies only when you are using Oracle's distributed functionality. This is the name of the database containing the object. The *dblink* qualifier lets you refer to an object in a database other than your local database. If you omit *dblink*, then Oracle assumes that you are referring to an object in your local database. Not all SQL statements allow you to access objects on remote databases.

You can include spaces around the periods separating the components of the reference to the object, but it is conventional to omit them.

How Oracle Resolves Schema Object References

When you refer to an object in a SQL statement, Oracle considers the context of the SQL statement and locates the object in the appropriate namespace. After locating the object, Oracle performs the statement's operation on the object. If the named object cannot be found in the appropriate namespace, then Oracle returns an error.

The following example illustrates how Oracle resolves references to objects within SQL statements. Consider this statement that adds a row of data to a table identified by the name `departments`:

```
INSERT INTO departments VALUES (
   280, 'ENTERTAINMENT_CLERK', 206, 1700);
```

Based on the context of the statement, Oracle determines that `departments` can be:

- A table in your own schema
- A view in your own schema
- A private synonym for a table or view
- A public synonym

Oracle always attempts to resolve an object reference within the namespaces in your own schema before considering namespaces outside your schema. In this example, Oracle attempts to resolve the name `dept` as follows:

1. First, Oracle attempts to locate the object in the namespace in your own schema containing tables, views, and private synonyms. If the object is a private synonym, then Oracle locates the object for which the synonym stands. This object could be in your own schema, another schema, or on another database. The object could also be another synonym, in which case Oracle locates the object for which this synonym stands.
2. If the object is in the namespace, then Oracle attempts to perform the statement on the object. In this example, Oracle attempts to add the row of data to `dept`. If the object is not of the correct type for the statement, then Oracle returns an error. In this example, `dept` must be a table, view, or a private synonym resolving to a table or view. If `dept` is a sequence, then Oracle returns an error.
3. If the object is not in any namespace searched in thus far, then Oracle searches the namespace containing public synonyms. If the object is in that namespace,

then Oracle attempts to perform the statement on it. If the object is not of the correct type for the statement, then Oracle returns an error. In this example, if `dept` is a public synonym for a sequence, then Oracle returns an error.

Note: If a public object type synonym has any dependent tables or user-defined types, then you cannot create an object with the same name as the synonym in the same schema as the dependent objects.

If the public object type synonym does not have any dependent tables or user-defined types, then you can create an object with the same name in the same schema as the dependent objects. Oracle invalidates any dependent objects and attempts to revalidate them when they are next accessed.

Referring to Objects in Other Schemas

To refer to objects in schemas other than your own, prefix the object name with the schema name:

```
schema.object
```

For example, this statement drops the `employees` table in the sample schema `hr`:

```
DROP TABLE hr.employees
```

Referring to Objects in Remote Databases

To refer to objects in databases other than your local database, follow the object name with the name of the database link to that database. A database link is a schema object that causes Oracle to connect to a remote database to access an object there. This section tells you:

- How to create database links
- How to use database links in your SQL statements

Creating Database Links

You create a database link with the statement CREATE DATABASE LINK on page 13-35. The statement lets you specify this information about the database link:

- The name of the database link
- The database connect string to access the remote database

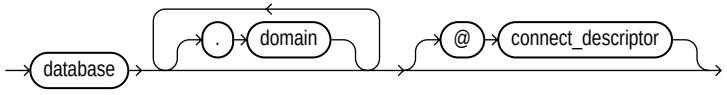
- The username and password to connect to the remote database

Oracle stores this information in the data dictionary.

Database Link Names When you create a database link, you must specify its name. Database link names are different from names of other types of objects. They can be as long as 128 bytes and can contain periods (.) and the "at" sign (@).

The name that you give to a database link must correspond to the name of the database to which the database link refers and the location of that database in the hierarchy of database names. The following syntax diagram shows the form of the name of a database link:

dblink::=



where:

- *database* should specify the *name* portion of the global name of the remote database to which the database link connects. This global name is stored in the data dictionary of the remote database; you can see this name in the GLOBAL_NAME view.
- *domain* should specify the *domain* portion of the global name of the remote database to which the database link connects. If you omit *domain* from the name of a database link, then Oracle qualifies the database link name with the domain of your local database as it currently exists in the data dictionary.
- *connect_descriptor* lets you further qualify a database link. Using connect descriptors, you can create multiple database links to the same database. For example, you can use connect descriptors to create multiple database links to different instances of the Real Application Clusters that access the same database.

The combination *database.domain* is sometimes called the "service name".

> **See Also:** *Oracle9i Net Services Administrator's Guide*

Username and Password Oracle uses the username and password to connect to the remote database. The username and password for a database link are optional.

Database Connect String The database connect string is the specification used by Oracle Net to access the remote database. For information on writing database connect strings, see the Oracle Net documentation for your specific network protocol. The database string for a database link is optional.

Referring to Database Links

Database links are available only if you are using Oracle's distributed functionality. When you issue a SQL statement that contains a database link, you can specify the database link name in one of these forms:

- *complete* is the complete database link name as stored in the data dictionary, including the *database*, *domain*, and optional *connect_descriptor* components.
- *partial* is the *database* and optional *connect_descriptor* components, but not the *domain* component.

Oracle performs these tasks before connecting to the remote database:

1. If the database link name specified in the statement is partial, then Oracle expands the name to contain the domain of the local database as found in the global database name stored in the data dictionary. (You can see the current global database name in the GLOBAL_NAME data dictionary view.)
2. Oracle first searches for a private database link in your own schema with the same name as the database link in the statement. Then, if necessary, it searches for a public database link with the same name.
   - Oracle always determines the username and password from the first matching database link (either private or public). If the first matching database link has an associated username and password, then Oracle uses it. If it does not have an associated username and password, then Oracle uses your current username and password.
   - If the first matching database link has an associated database string, then Oracle uses it. Otherwise Oracle searches for the next matching (public) database link. If no matching database link is found, or if no matching link has an associated database string, then Oracle returns an error.
3. Oracle uses the database string to access the remote database. After accessing the remote database, if the value of the GLOBAL_NAMES parameter is true, then Oracle verifies that the *database.domain* portion of the database link name matches the complete global name of the remote database. If this

condition is true, then Oracle proceeds with the connection, using the username and password chosen in Step 2. If not, Oracle returns an error.

4. If the connection using the database string, username, and password is successful, then Oracle attempts to access the specified object on the remote database using the rules for resolving object references and referring to objects in other schemas discussed earlier in this section.

You can disable the requirement that the *database.domain* portion of the database link name must match the complete global name of the remote database by setting to `false` the initialization parameter `GLOBAL_NAMES` or the `GLOBAL_NAMES` parameter of the `ALTER SYSTEM` or `ALTER SESSION` statement.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information on remote name resolution

Referencing Object Type Attributes and Methods

To reference object type attributes or methods in a SQL statement, you must fully qualify the reference with a table alias. Consider the following example from the sample schema `oe`, which contains a type `cust_address_typ` and a table `customers` with a `cust_address` column based on the `cust_address_typ`:

```
CREATE TYPE cust_address_typ AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
/
CREATE TABLE customers
  ( customer_id      NUMBER(6)
  , cust_first_name  VARCHAR2(20) CONSTRAINT cust_fname_nn NOT NULL
  , cust_last_name   VARCHAR2(20) CONSTRAINT cust_lname_nn NOT NULL
  , cust_address     cust_address_typ
  .
  .
  .
```

In a SQL statement, reference to the `postal_code` attribute must be fully qualified using a table alias, as illustrated in the following example:

```
SELECT c.cust_address.postal_code FROM customers c;

UPDATE customers c SET c.cust_address.postal_code = 'GU13 BE5'
   WHERE c.cust_address.city = 'Fleet';
```

To reference an object type's member method that does not accept arguments, you must provide "empty" parentheses. For example, the sample schema oe contains an object table catalogs_tab, based on catalog_typ, which contains the member function getCatalogName. In order to call this method in a SQL statement, you must provide empty parentheses as shows in this example:

```
SELECT c.getCatalogName() FROM catalogs_tab c
   WHERE category_id = 90;
```

> **See Also:** *Oracle9i Database Concepts* for more information on user-defined datatypes

3

Operators

An operator manipulates individual data items and returns a result.

This chapter contains these sections:

- About SQL Operators
- Arithmetic Operators
- Concatenation Operator
- Set Operators
- User-Defined Operators

This chapter discusses nonlogical (non-Boolean) operators. These operators cannot by themselves serve as the condition of a `WHERE` or `HAVING` clause in queries or subqueries. For information on logical operators, which serve as conditions, please refer to Chapter 5, "Conditions".

About SQL Operators

Operators manipulate individual data items called **operands** or **arguments**. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*).

Note: If you have installed Oracle Text, you can use the SCORE operator, which is part of that product, in Oracle Text queries. For more information on this operator, please refer to *Oracle Text Reference*.

Unary and Binary Operators

The two general classes of operators are:

- **unary**: A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

  ```
  operator operand
  ```

- **binary**: A binary operator operates on two operands. A binary operator appears with its operands in this format:

  ```
  operand1 operator operand2
  ```

Other operators with special formats accept more than two operands. If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Operator Precedence

Precedence is the order in which Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

Table 3–1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 3–1 SQL Operator Precedence

Operator	Operation
+, - (as unary operators), PRIOR	identity, negation, location in hierarchy
*, /	multiplication, division
+, - (as binary operators), ||	addition, subtraction, concatenation
SQL conditions are evaluated after SQL operators	See "Condition Precedence" on page 5-3

Precedence Example In the following expression, multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

```
1+2*3
```

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

SQL also supports set operators (UNION, UNION ALL, INTERSECT, and MINUS), which combine sets of rows returned by queries, rather than individual data items. All set operators have equal precedence.

See Also:

- "Set Operators" on page 3-6
- "Hierarchical Queries" on page 8-3 for information on the PRIOR operator, which is used only in hierarchical queries

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. Table 3–2 lists arithmetic operators.

Table 3–2 Arithmetic Operators

Operator	Purpose	Example
+ -	When these denote a positive or negative expression, they are unary operators.	`SELECT * FROM order_items WHERE quantity = -1;` `SELECT * FROM employees WHERE -salary < 0;`
	When they add or subtract, they are binary operators.	`SELECT hire_date FROM employees WHERE SYSDATE - hire_date > 365;`
* /	Multiply, divide. These are binary operators.	`UPDATE employees SET salary = salary * 1.1;`

Do not use two consecutive minus signs (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. The characters -- are used to begin comments within SQL statements. You should separate consecutive minus signs with a space or a parenthesis.

> **See Also:** "Comments" on page 2-90 for more information on comments within SQL statements

Concatenation Operator

The concatenation operator manipulates character strings and CLOB data. Table 3–3 describes the concatenation operator.

Table 3–3 Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings and CLOB data.	`SELECT 'Name is ' || last_name FROM employees;`

The result of concatenating two character strings is another character string. If both character strings are of datatype CHAR, the result has datatype CHAR and is limited to 2000 characters. If either string is of datatype VARCHAR2, the result has datatype VARCHAR2 and is limited to 4000 characters. If either argument is a CLOB, the result is a temporary CLOB. Trailing blanks in character strings are preserved by concatenation, regardless of the datatypes of the string or CLOB.

On most platforms, the concatenation operator is two solid vertical bars, as shown in Table 3–3. However, some IBM platforms use broken vertical bars for this operator. When moving SQL script files between systems having different character sets, such as between ASCII and EBCDIC, vertical bars might not be translated into the vertical bar required by the target Oracle environment. Oracle provides the CONCAT character function as an alternative to the vertical bar operator for cases when it is difficult or impossible to control translation performed by operating system or network utilities. Use this function in applications that will be moved between environments with differing character sets.

Although Oracle treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

See Also:

- "Character Datatypes" on page 2-9 for more information on the differences between the CHAR and VARCHAR2 datatypes
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for more information about CLOBs
- The functions CONCAT on page 6-33 and NVL on page 6-110

Example This example creates a table with both CHAR and VARCHAR2 columns, inserts values both with and without trailing blanks, and then selects these values and concatenates them. Note that for both CHAR and VARCHAR2 columns, the trailing blanks are preserved.

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(6),
        col3 VARCHAR2(6), col4 CHAR(6) );

INSERT INTO tab1 (col1,  col2,     col3,     col4)
        VALUES   ('abc', 'def   ', 'ghi   ', 'jkl');

SELECT col1||col2||col3||col4 "Concatenation"
        FROM tab1;

Concatenation
------------------------
abcdef   ghi   jkl
```

Set Operators

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table 3–4 lists SQL set operators. They are fully described, including restrictions on these operators, in "The UNION [ALL], INTERSECT, MINUS Operators" on page 8-6.

Table 3–4 Set Operators

Operator	Returns
`UNION`	All rows selected by either query
`UNION ALL`	All rows selected by either query, including all duplicates
`INTERSECT`	All distinct rows selected by both queries
`MINUS`	All distinct rows selected by the first query but not the second

User-Defined Operators

Like built-in operators, user-defined operators take a set of operands as input and return a result. However, you create them with the `CREATE OPERATOR` statement, and they are identified by names. They reside in the same namespace as tables, views, types, and standalone functions.

Once you have defined a new operator, you can use it in SQL statements like any other built-in operator. For example, you can use user-defined operators in the select list of a `SELECT` statement, the condition of a `WHERE` clause, or in `ORDER BY` clauses and `GROUP BY` clauses. However, you must have `EXECUTE` privilege on the operator to do so, because it is a user-defined object.

For example, if you define an operator `includes`, which takes as input a text column and a keyword and returns 1 if the row contains the specified keyword, you can then write the following SQL query:

```
SELECT * FROM product_descriptions
   WHERE includes (translated_description, 'Oracle and UNIX') = 1;
```

> **See Also:** CREATE OPERATOR on page 14-42 and *Oracle9i Data Cartridge Developer's Guide* for more information on user-defined operators

4

Expressions

This chapter describes how to combine values, operators, and functions into expressions.

This chapter includes these sections:

About SQL Expressions

An **expression** is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype NUMBER (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to CHAR datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in:

- The select list of the SELECT statement
- A condition of the WHERE clause and HAVING clause
- The CONNECT BY, START WITH, and ORDER BY clauses
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

For example, you could use an expression in place of the quoted string 'smith' in this UPDATE statement SET clause:

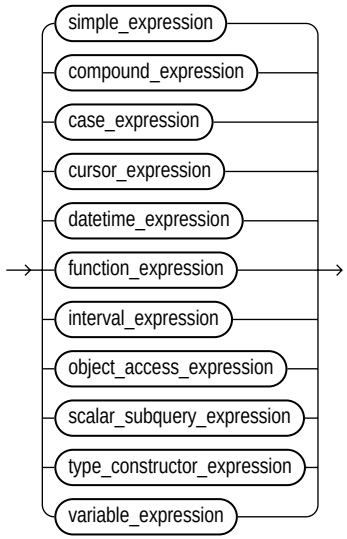
```
SET last_name = 'Smith';
```

This SET clause has the expression INITCAP(last_name) instead of the quoted string 'Smith':

```
SET last_name = INITCAP(last_name);
```

Expressions have several forms, as shown in the following syntax:

expr::=



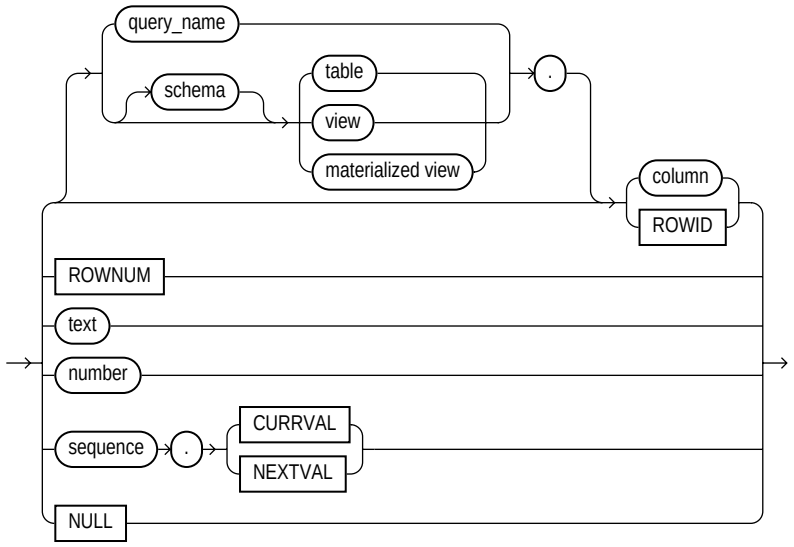
Oracle does not accept all forms of expressions in all parts of all SQL statements. You must use appropriate expression notation whenever *expr* appears in conditions, SQL functions, or SQL statements in other parts of this reference. The sections that follow describe and provide examples of the various forms of expressions.

> **See Also:** The individual SQL statements in Chapter 9 through Chapter 18 for information on restrictions on the expressions in that statement

Simple Expressions

A simple expression specifies column, pseudocolumn, constant, sequence number, or null.

simple_expression::=



In addition to the schema of a user, *schema* can also be "PUBLIC" (double quotation marks required), in which case it must qualify a public synonym for a table, view, or materialized view. Qualifying a public synonym with "PUBLIC" is supported only in data manipulation language (DML) statements, not data definition language (DDL) statements.

The *pseudocolumn* can be either LEVEL, ROWID, or ROWNUM. You can use a pseudocolumn only with a table, not with a view or materialized view. NCHAR and NVARCHAR2 are not valid pseudocolumn datatypes.

See Also:

- "Pseudocolumns" on page 2-83 for more information on pseudocolumns
- *subquery_factoring_clause* on page 18-10 for information on *query_name*

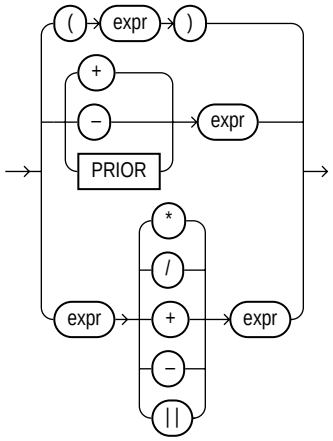
Some valid simple expressions are:

```
employees.last_name
'this is a text string'
10
N'this is an NCHAR string'
```

Compound Expressions

A compound expression specifies a combination of other expressions.

compound_expression::=



Note: You can use any built-in function as an expression ("Function Expressions" on page 4-11). However, in a compound expression, some combinations of functions are inappropriate and are rejected. For example, the `LENGTH` function is inappropriate within an aggregate function.

The `PRIOR` operator is used in `CONNECT BY` clauses of hierarchical queries.

See Also: "Operator Precedence" on page 3-2 and "Hierarchical Queries" on page 8-3

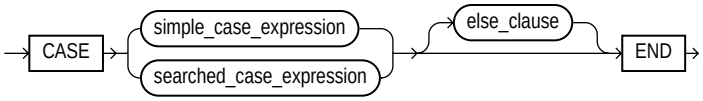
Some valid compound expressions are:

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate,'DD-MMM-YY'))
```

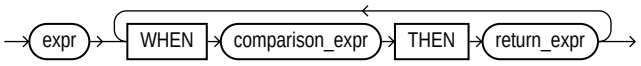
CASE Expressions

CASE expressions let you use IF ... THEN ... ELSE logic in SQL statements without having to invoke procedures. The syntax is:

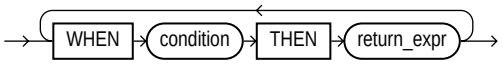
case_expression::=



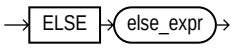
simple_case_expression::=



searched_case_expression::=



else_clause::=



In a simple CASE expression, Oracle searches for the first WHEN ... THEN pair for which *expr* is equal to *comparison_expr* and returns *return_expr*. If none of the WHEN ... THEN pairs meet this condition, and an ELSE clause exists, then Oracle returns *else_expr*. Otherwise, Oracle returns null. You cannot specify the literal NULL for all the *return_expr*s and the *else_expr*.

All of the expressions (*expr*, *comparison_expr*, and *return_expr*) must be of the same datatype, which can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

In a searched CASE expression, Oracle searches from left to right until it finds an occurrence of *condition* that is true, and then returns *return_expr*. If no *condition* is found to be true, and an ELSE clause exists, Oracle returns *else_expr*. Otherwise, Oracle returns null.

Note: The maximum number of arguments in a CASE expression is 255, and each WHEN ... THEN pair counts as two arguments. To avoid exceeding the limit of 128 choices, you can nest CASE expressions. That is *return_expr* can itself be a CASE expression.

See Also:

- COALESCE on page 6-31 and NULLIF on page 6-107 for alternative forms of CASE logic
- *Oracle9i Data Warehousing Guide* for examples using various forms of the CASE expression

Simple CASE Example For each customer in the sample `oe.customers` table, the following statement lists the credit limit as "Low" if it equals $100, "High" if it equals $5000, and "Medium" if it equals anything else.

```
SELECT cust_last_name,
   CASE credit_limit WHEN 100 THEN 'Low'
   WHEN 5000 THEN 'High'
   ELSE 'Medium' END
   FROM customers;

CUST_LAST_NAME       CASECR
-------------------- ------
...
Bogart               Medium
Nolte                Medium
Loren                Medium
Gueney               Medium
```

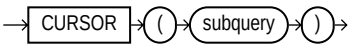
Searched CASE Example The following statement finds the average salary of the employees in the sample table `oe.employees`, using $2000 as the lowest salary possible:

```
SELECT AVG(CASE WHEN e.salary > 2000 THEN e.salary
   ELSE 2000 END) "Average Salary" from employees e;

Average Salary
--------------
    6461.68224
```

CURSOR Expressions

A CURSOR expression returns a nested cursor. This form of expression is equivalent to the PL/SQL REF CURSOR and can be passed as a REF CURSOR argument to a function.

cursor_expression::=



A nested cursor is implicitly opened when the cursor expression is evaluated. For example, if the cursor expression appears in a SELECT list, a nested cursor will be opened for each row fetched by the query. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is reexecuted
- The parent cursor is closed
- The parent cursor is cancelled
- An error arises during fetch on one of its parent cursors (it is closed as part of the clean-up)

Restrictions: The following restrictions apply to CURSOR expressions:

- If the enclosing statement is not a SELECT statement, nested cursors can appear only as REF CURSOR arguments of a procedure.
- If the enclosing statement is a SELECT statement, nested cursors can also appear in the outermost SELECT list of the query specification, or in the outermost SELECT list of another nested cursor.
- Nested cursors cannot appear in views.
- You cannot perform BIND and EXECUTE operations on nested cursors.

Examples The following example shows the use of a CURSOR expression in the select list of a query:

```
SELECT department_name, CURSOR(SELECT salary, commission_pct
   FROM employees e
   WHERE e.department_id = d.department_id)
   FROM departments d;
```

The next example shows the use of a CURSOR expression as a function argument. The example begins by creating a function in the sample OE schema that can accept the REF CURSOR argument. (The PL/SQL function body is shown in italics.)

```
CREATE FUNCTION f(cur SYS_REFCURSOR, mgr_hiredate DATE)
   RETURN NUMBER IS
   emp_hiredate DATE;
   before number :=0;
```

```
   after number:=0;
begin
  loop
    fetch cur into emp_hiredate;
    exit when cur%NOTFOUND;
    if emp_hiredate > mgr_hiredate then
      after:=after+1;
    else
      before:=before+1;
    end if;
  end loop;
  close cur;
  if before > after then
    return 1;
  else
    return 0;
  end if;
end;
/
```

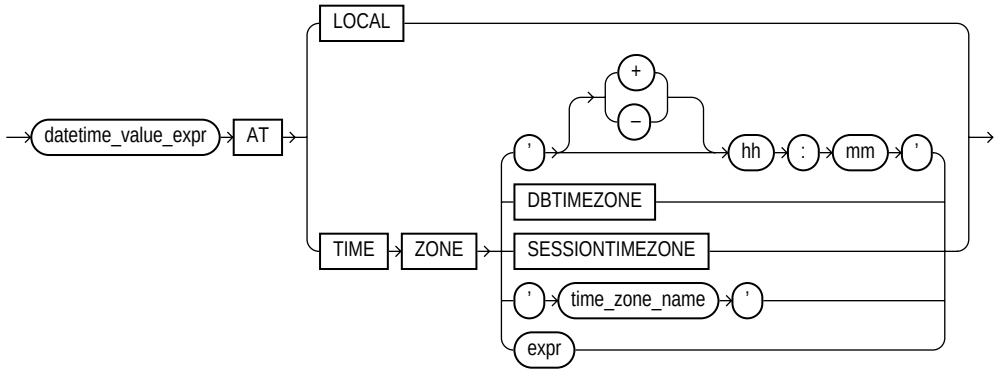
The function accepts a cursor and a date. The function expects the cursor to be a query returning a set of dates. The following query uses the function to find those managers in the sample `employees` table, most of whose employees were hired before the manager.

```
SELECT e1.last_name FROM employees e1
   WHERE f(
   CURSOR(SELECT e2.hire_date FROM employees e2
   WHERE e1.employee_id = e2.manager_id),
   e1.hire_date) = 1;

LAST_NAME
-------------------------
De Haan
Mourgos
Cambrault
Zlotkey
Higgens
```

Datetime Expressions

A datetime expression yields a value of one of the datetime datatypes.

datetime_expression::=



A *datetime_value_expr* can be a datetime column or a compound expression that yields a datetime value. Datetimes and intervals can be combined according to the rules defined in Table 2–5 on page 2-24. The three combinations that yield datetime values are valid in a datetime expression.

If you specify AT LOCAL, Oracle uses the current session time zone.

The settings for AT TIME ZONE are interpreted as follows:

- The string '(+|-)HH:MM' specifies a time zone as an offset from UTC.
- DBTIMEZONE: Oracle uses the database time zone established (explicitly or by default) during database creation.
- SESSIONTIMEZONE: Oracle uses the session time zone established by default or in the most recent ALTER SESSION statement.
- *time_zone_name*: Oracle returns the *datetime_value_expr* in the time zone indicated by *time_zone_name*. For a listing of valid time zone names, query the V$TIMEZONE_NAMES dynamic performance view.

  See Also: *Oracle9i Database Reference* for information on the dynamic performance views

- *expr*: If *expr* returns a character string with a valid time zone format, Oracle returns the input in that time zone. Otherwise, Oracle returns an error.

Example The following example converts the datetime value of one time zone to another time zone:

```
SELECT FROM_TZ(CAST(TO_DATE('1999-12-01 11:00:00',
```

```
      'YYYY-MM-DD HH:MI:SS') AS TIMESTAMP), 'America/New_York')
   AT TIME ZONE 'America/Los_Angeles' "West Coast Time"
   FROM DUAL;

West Coast Time
---------------------------------------------
01-DEC-99 08.00.00.000000 AM AMERICA/LOS_ANGELES
```

Function Expressions

You can use any built-in SQL function or user-defined function as an expression. Some valid built-in function expressions are:

```
LENGTH('BLAKE')
ROUND(1234.567*43)
SYSDATE
```

> **See Also:** "SQL Functions" on page 6-2 and "Aggregate Functions" on page 6-7 for information on built-in functions

A user-defined function expression specifies a call to:

- A function in an Oracle-supplied package (see *Oracle9i Supplied PL/SQL Packages and Types Reference*)
- A function in a user-defined package or type or in a standalone user-defined function (see "User-Defined Functions" on page 6-219)
- A user-defined function or operator (see CREATE OPERATOR on page 14-42, CREATE FUNCTION on page 13-49, and *Oracle9i Data Cartridge Developer's Guide*)

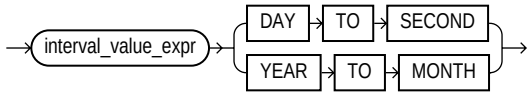
Some valid user-defined function expressions are:

```
circle_area(radius)
payroll.tax_rate(empno)
scott.payrol.tax_rate(dependents, empno)@ny
DBMS_LOB.getlength(column_name)
my_function(DISTINCT a_column)
```

INTERVAL Expressions

An interval expression yields a value of `INTERVAL YEAR TO MONTH` or `INTERVAL DAY TO SECOND`.

interval_expression::=



The *interval_value_expr* can be the value of an INTERVAL column or a compound expression that yields an interval value. Datetimes and intervals can be combined according to the rules defined in Table 2–5 on page 2-24. The six combinations that yield interval values are valid in an interval expression.

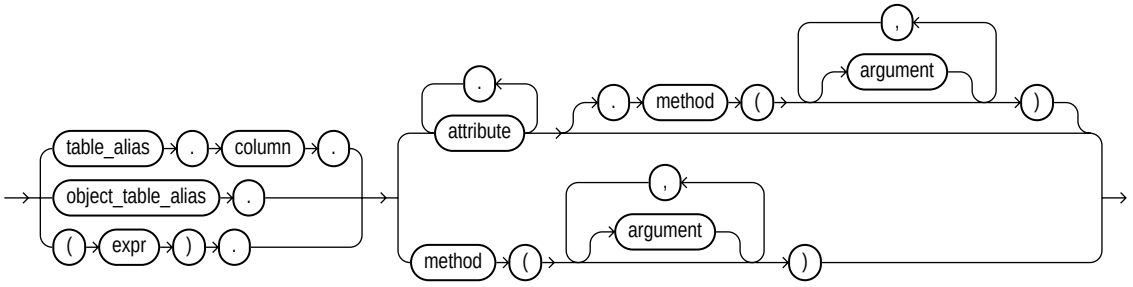
For example, the following statement subtracts the value of the order_date column in the sample table orders (a datetime value) from the system timestamp (another datetime value) to yield an interval value expression:

```
SELECT (SYSTIMESTAMP - order_date) DAY TO SECOND from orders;
```

Object Access Expressions

An object access expression specifies attribute reference and method invocation.

object_access_expression::=



The column parameter can be an object or REF column. If you specify *expr*, it must resolve to an object type.

When a type's member function is invoked in the context of a SQL statement, if the SELF argument is null, Oracle returns null and the function is not invoked.

Examples The following example creates a table based on the sample oe.order_item_typ object type, and then shows how you would update and select from the object column attributes.

```
CREATE TABLE short_orders (
```

```
   sales_rep VARCHAR2(25), item order_item_typ);

UPDATE short_orders s SET sales_rep = 'Unassigned';

SELECT o.item.line_item_id, o.item.quantity FROM short_orders o;
```

Scalar Subquery Expressions

A scalar subquery expression is a subquery that returns exactly one column value from one row. The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, then the value of the scalar subquery expression is NULL. If the subquery returns more than one row, then Oracle returns an error.

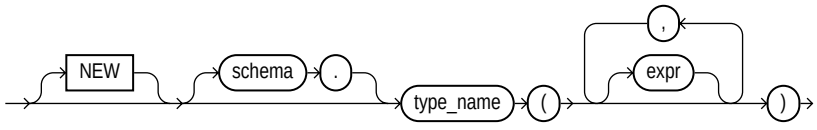
You can use a scalar subquery expression in most syntax that calls for an expression (*expr*). However, scalar subqueries are not valid expressions in the following places:

- As default values for columns
- As hash expressions for clusters
- In the RETURNING clause of DML statements
- As the basis of a function-based index
- In CHECK constraints
- In WHEN conditions of CASE expressions
- In GROUP BY and HAVING clauses
- In START WITH and CONNECT BY clauses
- In statements that are unrelated to queries, such as CREATE PROFILE

Type Constructor Expressions

A type constructor expression specifies a call to a type constructor. The argument to the type constructor is any expression.

type_constructor_expression::=



The NEW keyword instructs Oracle to construct a new object by invoking an appropriate constructor. The use of the NEW keyword is optional, but it is good practice to specify it.

If *type_name* is an **object type**, then the expressions must be an ordered list, where the first argument is a value whose type matches the first attribute of the object type, the second argument is a value whose type matches the second attribute of the object type, and so on. The total number of arguments to the constructor must match the total number of attributes of the object type.

If *type_name* is a **varray or nested table type**, then the expression list can contain zero or more arguments. Zero arguments implies construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

Type constructors can be invoked anywhere functions are invoked. They also have similar restrictions, such as a limit on the maximum number of arguments.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for additional information on type constructors

Expression Example This example uses the cust_address_typ type in the sample oe schema to show the use of an expression in the call to a type constructor (the PL/SQL is shown in italics):

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;
   DECLARE
      myaddr cust_address_typ := NEW cust_address_typ(
        '500 Oracle Parkway', 94065, 'Redwood Shores', 'CA','USA');
      alladdr address_book_t := NEW address_book_t();
   BEGIN
      INSERT INTO customers VALUES (
         666999, 'Smith', 'Joe', myaddr, NULL, NULL, NULL, NULL,
         NULL, NULL, NULL);
   END;
/
```

Subquery Example This example uses the warehouse_typ type in the sample schema oe to illustrate the use of a subquery in the call to the type constructor.

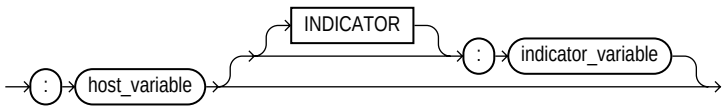
```
CREATE TABLE warehouse_tab OF warehouse_typ;

INSERT INTO warehouse_tab
   VALUES (warehouse_typ(101, 'new_wh', 201));
```

```
CREATE TYPE facility_typ AS OBJECT (
   facility_id NUMBER,
   warehouse_ref REF warehouse_typ);

CREATE TABLE buildings (b_id NUMBER, building facility_typ);

INSERT INTO buildings VALUES (10, facility_typ(102,
   (SELECT REF(w) FROM warehouse_tab w
      WHERE warehouse_name = 'new_wh')));

SELECT b.b_id, b.building.facility_id "FAC_ID",
   DEREF(b.building.warehouse_ref) "WH" FROM buildings b;

      B_ID     FAC_ID WH(WAREHOUSE_ID, WAREHOUSE_NAME, LOCATION_ID)
---------- ---------- ----------------------------------------
        10        102 WAREHOUSE_TYP(101, 'new_wh', 201)
```

Variable Expressions

A variable expression specifies a host variable with an optional indicator variable. This form of expression can appear only in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

variable_expression::=



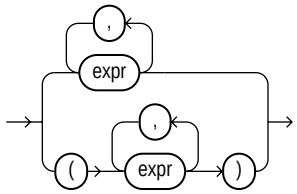
Some valid variable expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

Expression Lists

An expression list is a combination of other expressions.

expression_list::=



Expression lists can appear in comparison and membership conditions and in GROUP BY clauses of queries and subqueries.

Comparison and membership conditions appear in the conditions of WHERE clauses. They can contain *either* one or more comma-delimited expressions, or one or more sets of expressions where each set contains one or more comma-delimited expressions. In the latter case (multiple sets of expressions):

- Each set is bounded by parentheses
- Each set must contain the same number of expressions
- The number of expressions in each set must match the number of expressions before the operator in the comparison condition or before the IN keyword in the membership condition.

A comma-delimited list of expressions can contain no more than 1000 expressions. A comma-delimited list of sets of expressions can contain any number of sets, but each set can contain no more than 1000 expressions.

The following are some valid expression lists in conditions:

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
( ('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA') )
```

In the third example, the number of expressions in each set must equal the number of expressions in the first part of the condition. For example:

```
SELECT * FROM employees
  WHERE (first_name, last_name, email) IN
  (('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA'))
```

> **See Also:** "Comparison Conditions" on page 5-4 and "Membership Conditions" on page 5-9

In a simple GROUP BY clause, you can use either the upper or lower form of expression list:

```
SELECT department_id, MIN(salary), MAX(salary)
   FROM employees
   GROUP BY department_id, salary;

SELECT department_id, MIN(salary), MAX(salary)
   FROM employees
   GROUP BY (department_id, salary);
```

In ROLLUP, CUBE, and GROUPING SETS clauses of GROUP BY clauses, you can combine individual expressions with sets of expressions in the same expression list. The following example shows several valid grouping sets expression lists in one SQL statement:

```
SELECT
prod_category, prod_subcategory, country_id, cust_city, count(*)
   FROM  products, sales, customers
   WHERE sales.prod_id = products.prod_id
   AND sales.cust_id=customers.cust_id
   AND sales.time_id = '01-oct-00'
   AND customers.cust_year_of_birth BETWEEN 1960 and 1970
GROUP BY GROUPING SETS
  (
   (prod_category, prod_subcategory, country_id, cust_city),
   (prod_category, prod_subcategory, country_id),
   (prod_category, prod_subcategory),
    country_id
  );
```

See Also: SELECT on page 18-4

5
Conditions

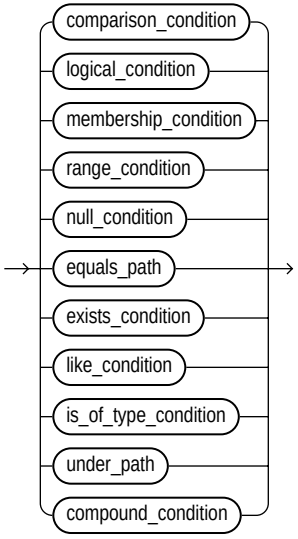
A condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of TRUE, FALSE, or unknown

This chapter contains the following sections:

About SQL Conditions

Conditions can have several forms, as shown in the following syntax.

condition::=



Note: If you have installed Oracle Text, then you can use the built-in conditions that are part of that product, including CONTAINS, CATSEARCH, and MATCHES. For more information on these Oracle Text elements, please refer to *Oracle Text Reference*.

The sections that follow describe the various forms of conditions. You must use appropriate condition syntax whenever *condition* appears in SQL statements.

You can use a condition in the WHERE clause of these statements:

- DELETE
- SELECT
- UPDATE

You can use a condition in any of these clauses of the SELECT statement:

- WHERE

- START WITH
- CONNECT BY
- HAVING

A condition could be said to be of the "logical" datatype, although Oracle does not formally support such a datatype.

The following simple condition always evaluates to TRUE:

```
1 = 1
```

The following more complex condition adds the sal value to the comm value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 2500:

```
NVL(salary, 0) + NVL(salary + (salary*commission_pct, 0) > 25000)
```

Logical conditions can combine multiple conditions into a single condition. For example, you can use the AND condition to combine two conditions:

```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
employees.department_id = departments.department_id
hire_date > '01-JAN-88'
job_id IN ('SA_MAN', 'SA_REP')
salary BETWEEN 5000 AND 10000
commission_pct IS NULL AND salary = 2100
```

> **See Also:** The description of each statement in Chapter 9 through Chapter 18 for the restrictions on the conditions in that statement

Condition Precedence

Precedence is the order in which Oracle evaluates different conditions in the same expression. When evaluating an expression containing multiple conditions, Oracle evaluates conditions with higher precedence before evaluating those with lower precedence. Oracle evaluates conditions with equal precedence from left to right within an expression.

Table 5–1 lists the levels of precedence among SQL condition from high to low. Conditions listed on the same line have the same precedence. As the table indicates, Oracle evaluates operators before conditions.

Table 5–1 SQL Condition Precedence

Type of Condition	Purpose
SQL operators are evaluated before SQL conditions	See "Operator Precedence" on page 3-2
=, !=, <, >, <=, >=,	comparison
IS [NOT] NULL, LIKE, [NOT] BETWEEN, [NOT] IN, EXISTS, IS OF *type*	comparison
NOT	exponentiation, logical negation
AND	conjunction
OR	disjunction

Comparison Conditions

Comparison conditions compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN.

Note: Large objects (LOBs) are not supported in comparison conditions. However, you can use PL/SQL programs for comparisons on CLOB data.

Table 5–2 lists comparison conditions.

Table 5–2 Comparison Conditions

Type of Condition	Purpose	Example
=	Equality test.	SELECT * FROM employees WHERE salary = 2500;
!= ^= <> ¬=	Inequality test. Some forms of the inequality condition may be unavailable on some platforms.	SELECT * FROM employees WHERE salary != 2500;

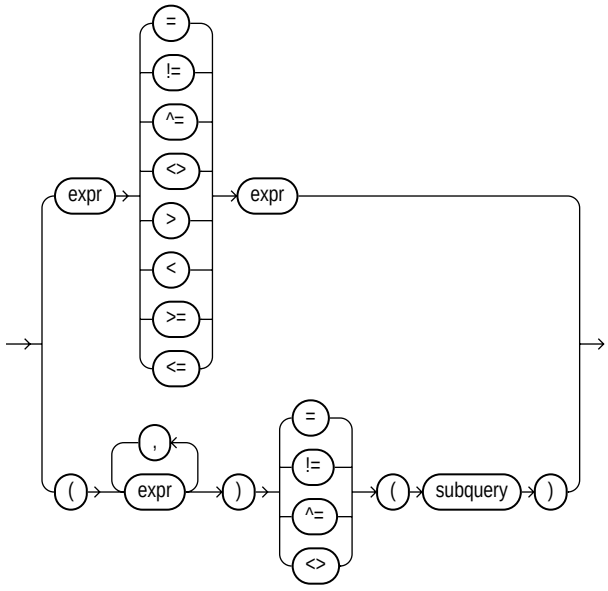
Table 5–2 (Cont.) Comparison Conditions

Type of Condition	Purpose	Example
>	"Greater than" and "less than" tests.	`SELECT * FROM employees WHERE salary > 2500;`
<		`SELECT * FROM employees WHERE salary < 2500;`
>=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM employees WHERE salary >= 2500;`
<=		`SELECT * FROM employees WHERE salary <= 2500;`
ANY SOME	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.	`SELECT * FROM employees WHERE salary = ANY (SELECT salary FROM employees WHERE department_id = 30);`
ALL	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to TRUE if the query returns no rows.	`SELECT * FROM employees WHERE salary >= ALL ( 1400, 3000);`

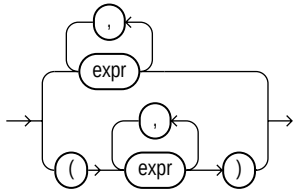
Simple Comparison Conditions

A simple comparison condition specifies a comparison with expressions or subquery results.

simple_comparison_condition::=



expression_list::=



If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of the *expression_list*, and the values returned by the subquery must match in number and datatype the expressions in *expression_list*.

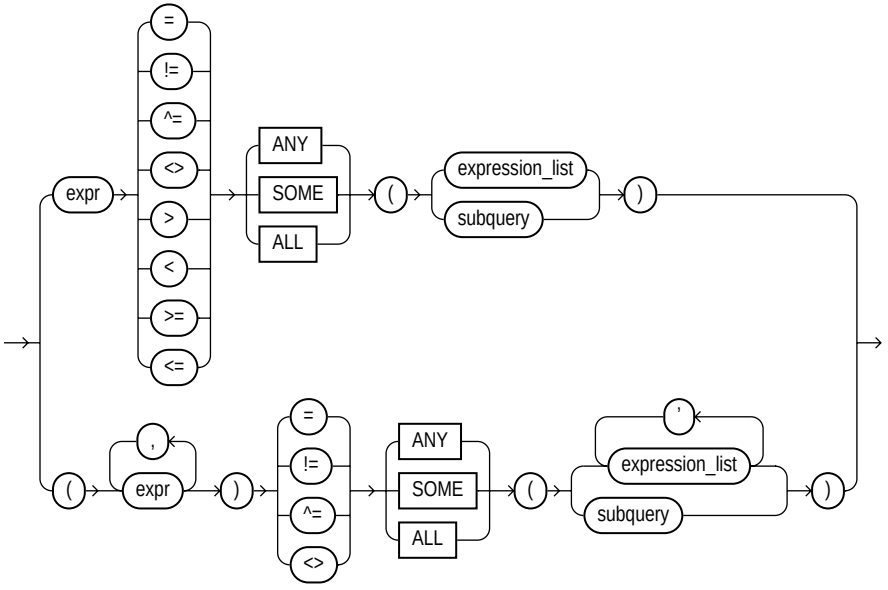
See Also:

- "Expression Lists" on page 4-15 for more information about combining expressions
- SELECT on page 18-4 for information about subqueries

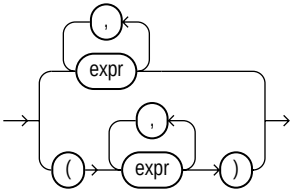
Group Comparison Conditions

A group comparison condition specifies a comparison with any or all members in a list or subquery.

group_comparison_condition::=



expression_list::=



If you use the upper form of this condition (with a single expression to the left of the operator), then you must use the upper form of *expression_list*. If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of *expression_list*, and the expressions in each *expression_list* must match in number and datatype the expressions to the left of the operator.

See Also:

- "Expression Lists" on page 4-15
- SELECT on page 18-4

Logical Conditions

A logical condition combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 5–3 lists logical conditions.

Table 5–3 Logical Conditions

Type of Condition	Operation	Examples
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, then it remains UNKNOWN.	`SELECT * FROM employees WHERE NOT (job_id IS NULL);` `SELECT * FROM employees WHERE NOT (salary BETWEEN 1000 AND 2000);`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	`SELECT * FROM employees WHERE job_id = 'PU_CLERK' AND department_id = 30;`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`SELECT * FROM employees WHERE job_id = 'PU_CLERK' OR department_id = 10;`

Table 5–4 shows the result of applying the NOT condition to an expression.

Table 5–4 NOT Truth Table

—	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

Table 5–5 shows the results of combining the AND condition to two expressions.

Table 5–5 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

For example, in the WHERE clause of the following SELECT statement, the AND logical condition is used to ensure that only those hired before 1984 and earning more than $1000 a month are returned:

```
SELECT * FROM employees
WHERE hire_date < TO_DATE('01-JAN-1989', 'DD-MON-YYYY')
   AND salary > 2500;
```

Table 5–6 shows the results of applying OR to two expressions.

Table 5–6 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

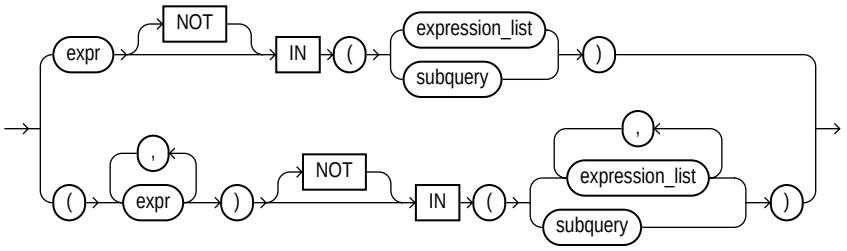
For example, the following query returns employees who have a 40% commission rate or a salary greater than $20,000:

```
SELECT employee_id FROM employees
   WHERE commission_pct = .4 OR salary > 20000;
```

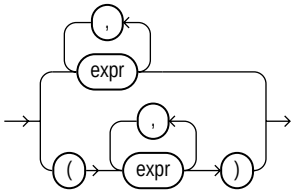
Membership Conditions

A membership condition tests for membership in a list or subquery.

membership_condition::=



expression_list::=



If you use the upper form of this condition (with a single expression to the left of the operator), then you must use the upper form of *expression_list*. If you use the lower form of this condition (with multiple expressions to the left of the operator), then you must use the lower form of *expression_list*, and the expressions in each *expression_list* must match in number and datatype the expressions to the left of the operator.

> **See Also:** "Expression Lists" on page 4-15

Table 5–7 lists the membership conditions.

Table 5–7 Membership Conditions

Type of Condition	Operation	Example

IN	"Equal to any member of" test. Equivalent to "= ANY".	`SELECT * FROM employees`
		`  WHERE job_id IN`
		`  ('PU_CLERK','SH_CLERK');`
		`SELECT * FROM employees`
		`  WHERE salary IN`
		`  (SELECT salary`
		`   FROM employees`
		`   WHERE department_id =30);`

NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	`SELECT * FROM employees`
		`  WHERE salary NOT IN`
		`  (SELECT salary`
		`   FROM employees`
		`  WHERE department_id = 30);`
		`SELECT * FROM employees`
		`  WHERE job_id NOT IN`
		`  ('PU_CLERK', 'SH_CLERK');`

If any item in the list following a NOT IN operation evaluates to null, then all rows evaluate to FALSE or UNKNOWN, and no rows are returned. For example, the following statement returns the string 'TRUE' for each row:

```
SELECT 'True' FROM employees
   WHERE department_id NOT IN (10, 20);
```

However, the following statement returns no rows:

```
SELECT 'True' FROM employees
    WHERE department_id NOT IN (10, 20, NULL);
```

The preceding example returns no rows because the WHERE clause condition evaluates to:

```
department_id != 10 AND department_id != 20 AND department_id != null
```

Because the third condition compares department_id with a null, it results in an UNKNOWN, so the entire expression results in FALSE (for rows with department_id equal to 10 or 20). This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

Moreover, if a NOT IN condition references a subquery that returns no rows at all, then all rows will be returned, as shown in the following example:

```
SELECT 'True' FROM employees
   WHERE department_id NOT IN (SELECT 0 FROM dual WHERE 1=2);
```

Restriction on LEVEL in WHERE clauses: In a [NOT] IN condition in a WHERE clause, if the right-hand side of the condition is a subquery, you cannot use LEVEL on the left-hand side of the condition. However, you can specify LEVEL in a subquery of the FROM clause to achieve the same result. For example, the following statement is not valid:

```
SELECT employee_id, last_name FROM employees
   WHERE (employee_id, LEVEL)
      IN (SELECT employee_id, 2 FROM employees)
   START WITH employee_id = 2
   CONNECT BY PRIOR employee_id = manager_id;
```

But the following statement is valid because it encapsulates the query containing the LEVEL information in the FROM clause:

```
SELECT v.employee_id, v.last_name, v.lev
   FROM
      (SELECT employee_id, last_name, LEVEL lev
      FROM employees v
      START WITH employee_id = 100
      CONNECT BY PRIOR employee_id = manager_id) v
   WHERE (v.employee_id, v.lev) IN
      (SELECT employee_id, 2 FROM employees);
```

Range Conditions

A range condition tests for inclusion in a range.

range_condition::=

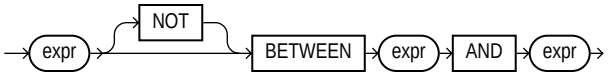


Table 5–8 describes the range conditions.

Table 5–8 Range Conditions

Type of Condition	Operation	Example
`[NOT] BETWEEN x AND y`	[Not] greater than or equal to *x* and less than or equal to *y*.	`SELECT * FROM employees WHERE salary BETWEEN 2000 AND 3000;`

Null Conditions

A NULL condition tests for nulls.

null_condition::=

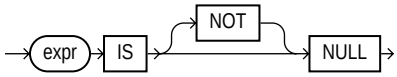


Table 5–9 lists the null conditions.

Table 5–9 Null Conditions

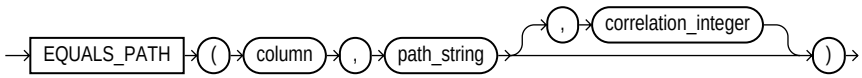
Type of Condition	Operation	Example
`IS [NOT] NULL`	Tests for nulls. This is the only condition that you should use to test for nulls. **See Also:** "Nulls" on page 2-81	`SELECT last_name FROM employees WHERE commission_pct IS NULL;`

EQUALS_PATH

The EQUALS_PATH condition determines whether a resource in the Oracle XML database can be found in the database at a specified path.

Use this condition in queries to RESOURCE_VIEW and PATH_VIEW. These public views provide a mechanism for SQL access to data stored in the XML database repository. RESOURCE_VIEW contains one row for each resource in the repository, and PATH_VIEW contains one row for each unique path in the repository.

equals_path::=



This condition applies only to the path as specified. It is similar to but more restrictive than UNDER_PATH.

The optional *correlation_number* argument correlates the EQUALS_PATH condition with its ancillary functions PATH and DEPTH.

> **See Also:** UNDER_PATH on page 5-20, DEPTH on page 6-55, and PATH on page 6-112

Example

The view RESOURCE_VIEW computes the paths (in the any_path column) that lead to all XML resources (in the res column) in the database repository. The following example queries the RESOURCE_VIEW view to find the paths to the resources in the sample schema oe. The EQUALS_PATH condition causes the query to return only the specified path:

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE EQUALS_PATH(res, '/sys/schemas/OE/www.oracle.com')=1;

ANY_PATH
--------------------------------------------
/sys/schemas/OE/www.oracle.com
```

Compare this example with that for UNDER_PATH on page 5-20.

EXISTS Conditions

An EXISTS condition tests for existence of rows in a subquery.

exists_condition::=

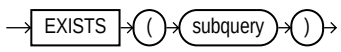


Table 5–10 shows the EXISTS condition.

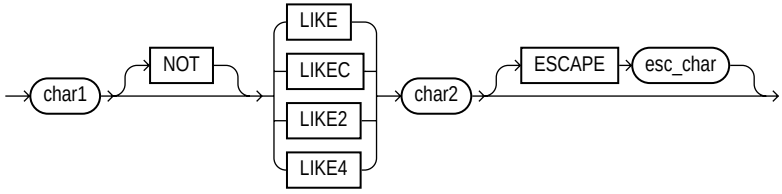
Table 5–10 EXISTS Condition

Type of Condition	Operation	Example
EXISTS	TRUE if a subquery returns at least one row.	`SELECT department_id FROM departments d WHERE EXISTS (SELECT * FROM employees e WHERE d.department_id = e.department_id);`

LIKE Conditions

The LIKE conditions specify a test involving pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the LIKE conditions match a portion of one character value to another by searching the first value for the pattern specified by the second. LIKE calculates strings using characters as defined by the input character set. LIKEC uses Unicode complete characters. LIKE2 uses UCS2 codepoints. LIKE4 uses USC4 codepoints.

like_condition::=



In this syntax:

- *char1* is a character expression, such as a character column, called the **search value**.
- *char2* is a character expression, usually a literal, called the **pattern**.
- *esc_char* is a character expression, usually a literal, called the **escape character**.

If *esc_char* is not specified, then there is no default escape character. If any of *char1*, *char2*, or *esc_char* is null, then the result is unknown. Otherwise, the escape character, if specified, must be a character string of length 1.

All of the character expressions (*char1*, *char2*, and *esc_char*) can be of any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. If they differ, then Oracle converts all of them to the datatype of *char1*.

The pattern can contain the special pattern-matching characters:

- % matches any string of any length (including length 0)
- _ matches any single character.

To search for the characters % and _, precede them by the escape character. For example, if the escape character is @, then you can use @% to search for %, and @_ to search for _.

To search for the escape character, repeat it. For example, if @ is the escape character, then you can use @@ to search for @.

In the pattern, the escape character should be followed by one of %, _, or the escape character itself.

Table 5–11 describes the LIKE conditions.

Table 5–11 LIKE Conditions

Type of Condition	Operation	Example
`x [NOT] LIKE y [ESCAPE 'z']`	TRUE if *x* does [not] match the pattern *y*. Within *y*, the character "%" matches any string of zero or more characters except null. The character "_" matches any single character. Any character can follow ESCAPE except percent (%) and underbar (_). A wildcard character is treated as a literal if preceded by the character designated as the escape character.	`SELECT last_name FROM employees WHERE last_name LIKE '%A\_B%' ESCAPE '\';`

To process the LIKE conditions, Oracle divides the pattern into subpatterns consisting of one or two characters each. The two-character subpatterns begin with the escape character and the other character is %, or _, or the escape character.

Let P_1, P_2, ..., P_n be these subpatterns. The like condition is true if there is a way to partition the search value into substrings S_1, S_2, ..., S_n so that for all *i* between 1 and *n*:

- If P_i is _, then S_i is a single character.
- If P_i is %, then S_i is any string.
- If P_i is two characters beginning with an escape character, then S_i is the second character of P_i.
- Otherwise, $P_i = S_i$.

With the LIKE conditions, you can compare a value to a pattern rather than to a constant. The pattern must appear after the LIKE keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with 'R':

```
SELECT salary
    FROM employees
    WHERE last_name LIKE 'R%';
```

The following query uses the = operator, rather than the LIKE condition, to find the salaries of all employees with the name 'R%':

```
SELECT salary
    FROM employees
    WHERE last_name = 'R%';
```

The following query finds the salaries of all employees with the name 'SM%'. Oracle interprets 'SM%' as a text literal, rather than as a pattern, because it precedes the LIKE keyword:

```
SELECT salary
    FROM employees
    WHERE 'SM%' LIKE last_name;
```

Patterns typically use special characters that Oracle matches with different characters in the value:

- An underscore (_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
- A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. Note that the pattern '%' cannot match a null.

Case Sensitivity

Case is significant in all conditions comparing character expressions including the LIKE condition and the equality (=) operators. You can use the UPPER function to perform a case-insensitive match, as in this condition:

```
UPPER(last_name) LIKE 'SM%'
```

Pattern Matching on Indexed Columns

When you use LIKE to search an indexed column for a pattern, Oracle can use the index to improve the statement's performance if the leading character in the pattern is not "%" or "_". In this case, Oracle can scan the index by this leading character. If the first character in the pattern is "%" or "_", then the index cannot improve the query's performance because Oracle cannot scan the index.

General Examples

This condition is true for all last_name values beginning with "Ma":

```
last_name LIKE 'Ma%'
```

All of these last_name values make the condition true:

```
Mallin, Markle, Marlow, Marvins, Marvis, Matos
```

Case is significant, so last_name values beginning with "MA", "ma", and "mA" make the condition false.

Consider this condition:

```
last_name LIKE 'SMITH_'
```

This condition is true for these last_name values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for 'SMITH', since the special character "_" must match exactly one character of the lastname value.

ESCAPE Clause Example

You can include the actual characters "%" or "_" in the pattern by using the ESCAPE clause, which identifies the escape character. If the escape character appears in the pattern before the character "%" or "_" then Oracle interprets this character literally in the pattern, rather than as a special pattern matching character.

To search for employees with the pattern 'A_B' in their name:

```
SELECT last_name
    FROM employees
    WHERE last_name LIKE '%A\_B%' ESCAPE '\';
```

The ESCAPE clause identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

Patterns Without % Example

If a pattern does not contain the "%" character, then the condition can be true only if both operands have the same length. Consider the definition of this table and the values inserted into it:

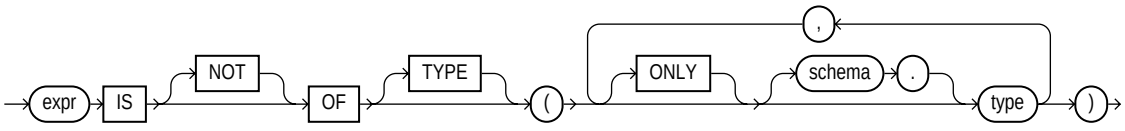
```
CREATE TABLE ducks (f CHAR(6), v VARCHAR2(6));
INSERT INTO ducks VALUES ('DUCK', 'DUCK');
SELECT '*'||f||'*' "char",
   '*'||v||'*' "varchar"
   FROM ducks;

char     varchar
-------- --------
*DUCK  * *DUCK*
```

Because Oracle blank-pads CHAR values, the value of f is blank-padded to 6 bytes. v is not blank-padded and has length 4.

IS OF *type* Conditions

Use the IS OF *type* condition to test object instances based on their specific type information.

is_of_type_condition::=



You must have EXECUTE privilege on all types referenced by *type*, and all *type*s must belong to the same type family.

This condition evaluates to null if *expr* is null. If *expr* is not null, then the condition evaluates to true (or false if you specify the NOT keyword) under either of these circumstances:

- The most specific type of *expr* is the subtype of one of the types specified in the *type* list and you have not specified ONLY for the type, or
- The most specific type of *expr* is explicitly specified in the *type* list.

The *expr* frequently takes the form of the VALUE function with a correlation variable.

The following example uses the sample table oe.persons, which is built on a type hierarchy in "Substitutable Table and Column Examples" on page 15-67. The example uses the IS OF *type* condition to restrict the query to specific subtypes:

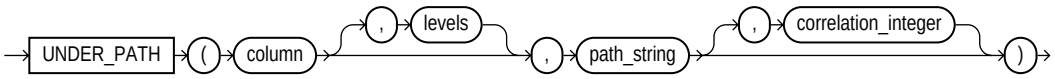
```
SELECT * FROM persons p
   WHERE VALUE(p) IS OF TYPE (employee_t);

NAME                          SSN
---------------------------
Joe                         32456
Tim                          5678

SELECT * FROM persons p
   WHERE VALUE(p) IS OF (ONLY part_time_emp_t);

NAME                          SSN
---------------------------
Tim                         5678
```

UNDER_PATH

The UNDER_PATH condition determines whether resources specified in a column can be found under a particular path specified by *path_string* in the Oracle XML database repository. The path information is computed by the RESOURCE_VIEW view, which you query to use this condition.

Use this condition in queries to RESOURCE_VIEW and PATH_VIEW. These public views provide a mechanism for SQL access to data stored in the XML database repository. RESOURCE_VIEW contains one row for each resource in the repository, and PATH_VIEW contains one row for each unique path in the repository.

under_path::=



The optional *levels* argument indicates the number of levels down from *path_string* Oracle should search. Oracle treats values less than 0 as 0.

The optional *correlation_integer* argument correlates the UNDER_PATH condition with its ancillary functions PATH and DEPTH.

See Also:

- The related condition EQUALS_PATH on page 5-13
- The ancillary functions DEPTH on page 6-55 and PATH on page 6-112

Example

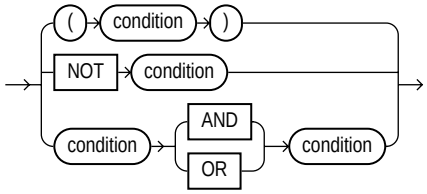
The view RESOURCE_VIEW computes the paths (in the any_path column) that lead to all XML resources (in the res column) in the database repository. The following example queries the RESOURCE_VIEW view to find the paths to the resources in the sample schema oe. The query returns the path of the XML schema that was created in "XMLType Table Examples" on page 15-71:

```
SELECT ANY_PATH FROM RESOURCE_VIEW
   WHERE UNDER_PATH(res, '/sys/schemas/OE/www.oracle.com')=1;

ANY_PATH
--------------------------------------------
/sys/schemas/OE/www.oracle.com/xwarehouses.xsd
```

Compound Conditions

A compound condition specifies a combination of other conditions.

compound_condition::=



See Also: "Logical Conditions" on page 5-8 for more information about NOT, AND, and OR conditions

6

Functions

Functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

This chapter contains these sections:

SQL Functions

SQL functions are built into Oracle and are available for use in various appropriate SQL statements. Do not confuse SQL functions with user functions written in PL/SQL.

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, then Oracle implicitly converts the argument to the expected datatype before performing the SQL function. If you call a SQL function with a null argument, then the SQL function automatically returns null. The only SQL functions that do not necessarily follow this behavior are CONCAT, NVL, and REPLACE.

In the syntax diagrams for SQL functions, arguments are indicated by their datatypes. When the parameter "function" appears in SQL syntax, replace it with one of the functions described in this section. Functions are grouped by the datatypes of their arguments and their return values.

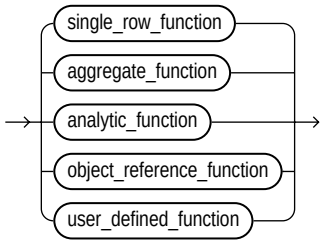
Note: When you apply SQL functions to LOB columns, Oracle creates temporary LOBs during SQL and PL/SQL processing. You should ensure that temporary tablespace quota is sufficient for storing these temporary LOBs for your application.

See Also:

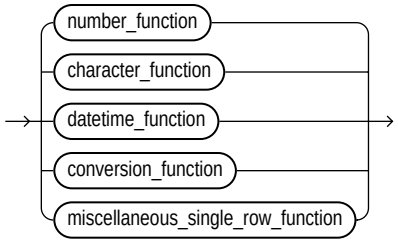
- "User-Defined Functions" on page 6-219 for information on user functions
- *Oracle Text Reference* for information on functions used with Oracle Text
- "Data Conversion" on page 2-48 for implicit conversion of datatypes

The syntax showing the categories of functions follows:

function::=



single_row_function::=



The sections that follow list the built-in SQL functions in each of the groups illustrated in the preceding diagrams except user-defined functions. All of the built-in SQL functions are then described in alphabetical order. User-defined functions are described at the end of this chapter.

Single-Row Functions

Single-row functions return a single result row for every row of a queried table or view. These functions can appear in select lists, `WHERE` clauses, `START WITH` and `CONNECT BY` clauses, and `HAVING` clauses.

Number Functions

Number functions accept numeric input and return numeric values. Most of these functions return values that are accurate to 38 decimal digits. The transcendental functions `COS`, `COSH`, `EXP`, `LN`, `LOG`, `SIN`, `SINH`, `SQRT`, `TAN`, and `TANH` are accurate to 36 decimal digits. The transcendental functions `ACOS`, `ASIN`, `ATAN`, and `ATAN2` are accurate to 30 decimal digits. The number functions are:

ABS	COSH	SIGN
ACOS	EXP	SIN
ASIN	FLOOR	SINH
ATAN	LN	SQRT
ATAN2	LOG	TAN
BITAND	MOD	TANH
CEIL	POWER	TRUNC (number)
COS	ROUND (number)	WIDTH_BUCKET

Character Functions Returning Character Values

Character functions that return character values return values of the same datatype as the input argument.

- Functions that return CHAR values are limited in length to 2000 bytes.
- Functions that return VARCHAR2 values are limited in length to 4000 bytes.

  For both of these types of functions, if the length of the return value exceeds the limit, then Oracle truncates it and returns the result without an error message.
- Functions that return CLOB values are limited to 4 GB.

  For CLOB functions, if the length of the return values exceeds the limit, then Oracle raises an error and returns no data.

The character functions that return character values are:

CHR	NLS_LOWER	SUBSTR
CONCAT	NLSSORT	TRANSLATE
INITCAP	NLS_UPPER	TREAT
LOWER	REPLACE	TRIM
LPAD	RPAD	UPPER
LTRIM	RTRIM	
NLS_INITCAP	SOUNDEX	

Character Functions Returning Number Values

Character functions that return number values can take as their argument any character datatype.

The character functions that return number values are:

ASCII	INSTR	LENGTH

Datetime Functions

Datetime functions operate on values of the DATE datatype. All datetime functions return a datetime or interval value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number. The datetime functions are:

ADD_MONTHS	MONTHS_BETWEEN	SYSDATE
CURRENT_DATE	NEW_TIME	SYSTIMESTAMP
CURRENT_TIMESTAMP	NEXT_DAY	TO_DSINTERVAL
DBTIMEZONE	NUMTODSINTERVAL	TO_TIMESTAMP
EXTRACT (datetime)	NUMTOYMINTERVAL	TO_TIMESTAMP_TZ
FROM_TZ	ROUND (date)	TO_YMINTERVAL
LAST_DAY	SESSIONTIMEZONE	TRUNC (date)
LOCALTIMESTAMP	SYS_EXTRACT_UTC	TZ_OFFSET

Conversion Functions

Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention *datatype* TO *datatype*. The first datatype is the input datatype. The second datatype is the output datatype. The SQL conversion functions are:

ASCIISTR
BIN_TO_NUM
CAST
CHARTOROWID
COMPOSE
CONVERT
DECOMPOSE
HEXTORAW
NUMTODSINTERVAL
NUMTOYMINTERVAL
RAWTOHEX
RAWTONHEX
ROWIDTOCHAR
ROWIDTONCHAR
TO_CHAR (character)
TO_CHAR (datetime)
TO_CHAR (number)
TO_CLOB
TO_DATE
TO_DSINTERVAL
TO_LOB
TO_MULTI_BYTE
TO_NCHAR (character)
TO_NCHAR (datetime)
TO_NCHAR (number)
TO_NCLOB
TO_NUMBER
TO_SINGLE_BYTE
TO_YMINTERVAL
TRANSLATE ... USING
UNISTR

Miscellaneous Single-Row Functions

The following single-row functions do not fall into any of the other single-row function categories:

BFILENAME
COALESCE
DECODE
DEPTH
DUMP
EMPTY_BLOB, EMPTY_CLOB
EXISTSNODE
EXTRACT (XML)
EXTRACTVALUE
GREATEST
LEAST
NLS_CHARSET_DECL_LEN
NLS_CHARSET_ID
NLS_CHARSET_NAME
NULLIF
NVL
NVL2
PATH
SYS_CONNECT_BY_PATH
SYS_CONTEXT
SYS_DBURIGEN
SYS_EXTRACT_UTC
SYS_GUID
SYS_TYPEID
SYS_XMLAGG
SYS_XMLGEN
UID
UPDATEXML
USER
USERENV
VSIZE
XMLAGG
XMLCOLATTVAL
XMLCONCAT
XMLFOREST
XMLSEQUENCE
XMLTRANSFORM

Aggregate Functions

Aggregate functions return a single result row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where Oracle divides the rows of a queried table or view into groups. In a query containing a GROUP BY clause, the elements of the select list can be aggregate functions, GROUP BY expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group.

If you omit the GROUP BY clause, then Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the HAVING clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

> **See Also:** "Using the GROUP BY Clause: Examples" on page 18-30 and the "HAVING Clause" on page 18-23 for more information on the GROUP BY clause and HAVING clauses in queries and subqueries

Many (but not all) aggregate functions that take a single argument accept these clauses:

- DISTINCT causes an aggregate function to consider only distinct values of the argument expression.
- ALL causes an aggregate function to consider all values, including all duplicates.

For example, the DISTINCT average of 1, 1, 1, and 3 is 2. The ALL average is 1.5. If you specify neither, then the default is ALL.

All aggregate functions except COUNT(*) and GROUPING ignore nulls. You can use the NVL function in the argument to an aggregate function to substitute a value for a null. COUNT never returns null, but returns either a number or zero. For all the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null.

You can nest aggregate functions. For example, the following example calculates the average of the maximum salaries of all the departments in the sample schema hr:

```
SELECT AVG(MAX(salary)) FROM employees GROUP BY department_id;

AVG(MAX(SALARY))
----------------
           10925
```

This calculation evaluates the inner aggregate (MAX(salary)) for each group defined by the GROUP BY clause (department_id), and aggregates the results again.

The aggregate functions are:

AVG
CORR
COUNT
COVAR_POP
COVAR_SAMP
CUME_DIST
DENSE_RANK
FIRST
GROUP_ID
GROUPING
GROUPING_ID
LAST
MAX
MIN
PERCENTILE_CONT
PERCENTILE_DISC
PERCENT_RANK
RANK
REGR_ (Linear Regression) Functions
STDDEV
STDDEV_POP
STDDEV_SAMP
SUM
VAR_POP
VAR_SAMP
VARIANCE

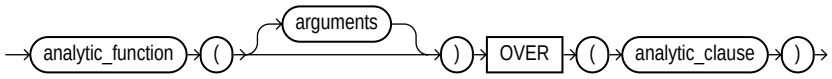
Analytic Functions

Analytic functions compute an aggregate value based on a group of rows. They differ from aggregate functions in that they return multiple rows for each group. The group of rows is called a **window** and is defined by the analytic clause. For each row, a "sliding" window of rows is defined. The window determines the range of rows used to perform the calculations for the "current row". Window sizes can be based on either a physical number of rows or a logical interval such as time.

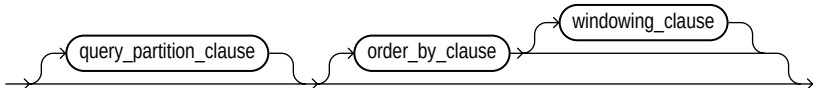
Analytic functions are the last set of operations performed in a query except for the final ORDER BY clause. All joins and all WHERE, GROUP BY, and HAVING clauses are completed before the analytic functions are processed. Therefore, analytic functions can appear only in the select list or ORDER BY clause.

Analytic functions are commonly used to compute cumulative, moving, centered, and reporting aggregates.

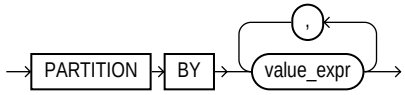
analytic_function::=



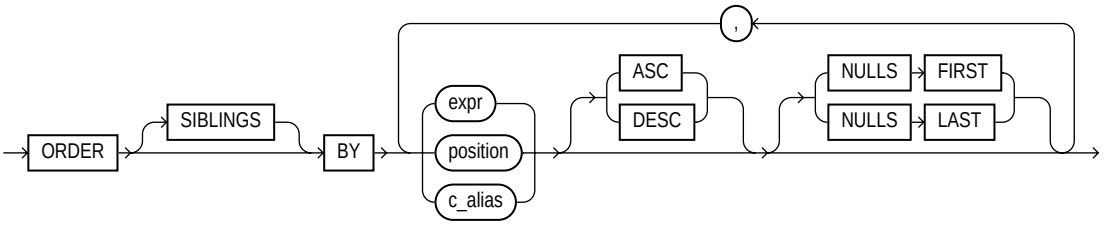
analytic_clause::=



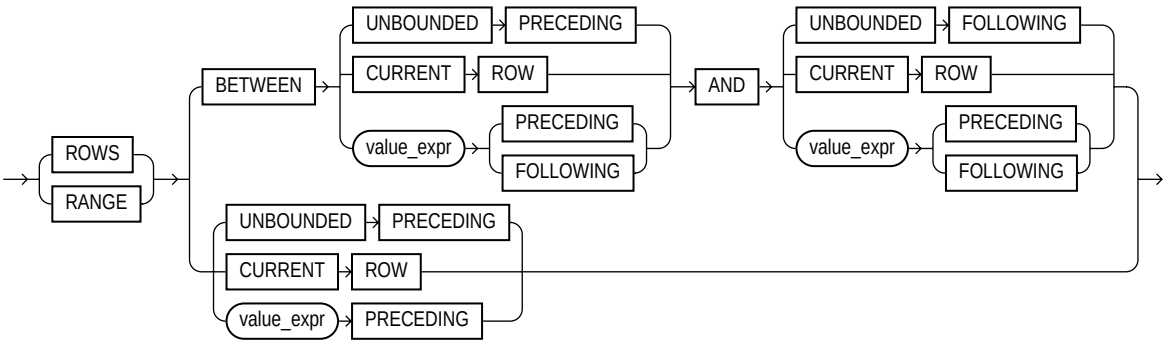
query_partition_clause::=



order_by_clause::=



windowing_clause::=



The keywords and parameters of this syntax are discussed in the sections that follow.

analytic_function

Specify the name of an analytic function (see the listing of analytic functions following this discussion of keywords and parameters).

arguments

Analytic functions take 0 to 3 arguments.

analytic_clause

Use OVER *analytic_clause* to indicate that the function operates on a query result set. That is, it is computed after the FROM, WHERE, GROUP BY, and HAVING clauses. You can specify analytic functions with this clause in the select list or ORDER BY clause. To filter the results of a query based on an analytic function, nest these functions within the parent query, and then filter the results of the nested subquery.

Notes:

- You cannot specify any analytic function in any part of the *analytic_clause*. That is, you cannot nest analytic functions. However, you can specify an analytic function in a subquery and compute another analytic function over it.
- You can specify OVER *analytic_clause* with user-defined analytic functions as well as built-in analytic functions. See CREATE FUNCTION on page 13-49.

query_partition_clause

Use the PARTITION BY clause to partition the query result set into groups based on one or more *value_expr*. If you omit this clause, then the function treats all rows of the query result set as a single group.

You can specify multiple analytic functions in the same query, each with the same or different PARTITION BY keys.

Note: If the objects being queried have the parallel attribute, and if you specify an analytic function with the *query_partition_clause*, then the function computations are parallelized as well.

Valid values of *value_expr* are constants, columns, nonanalytic functions, function expressions, or expressions involving any of these.

order_by_clause

Use the *order_by_clause* to specify how data is ordered within a partition. For all analytic functions except PERCENTILE_CONT and PERCENTILE_DISC (which take only a single key), you can order the values in a partition on multiple keys, each defined by a *value_expr* and each qualified by an ordering sequence.

Within each function, you can specify multiple ordering expressions. Doing so is especially useful when using functions that rank values, because the second expression can resolve ties between identical values for the first expression.

Note: Whenever the *order_by_clause* results in identical values for multiple rows, the function returns the same result for each of those rows. Please refer to the analytic example for SUM on page 6-151 for an illustration of this behavior.

Restriction: When used in an analytic function, the *order_by_clause* must take an expression (*expr*). The SIBLINGS keyword is not valid (it is relevant only in hierarchical queries). Position (*position*) and column aliases (*c_alias*) are invalid. Otherwise this *order_by_clause* is the same as that used to order the overall query or subquery.

ASC | DESC Specify the ordering sequence (ascending or descending). ASC is the default.

NULLS FIRST | NULLS LAST Specify whether returned rows containing nulls should appear first or last in the ordering sequence.

NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.

Note: Analytic functions always operate on rows in the order specified in the *order_by_clause* of the function. However, the *order_by_clause* of the function does not guarantee the order of the result. Use the *order_by_clause* of the query to guarantee the final result ordering.

See Also: *order_by_clause* of SELECT on page 18-25 for more information on this clause

windowing_clause

Some analytic functions allow the *windowing_clause*. In the listing of analytic functions at the end of this section, the functions that allow the *windowing_clause* are followed by an asterisk (*).

ROWS | RANGE These keywords define for each row a "window" (a physical or logical set of rows) used for calculating the function result. The function is then applied to all the rows in the window. The window "slides" through the query result set or partition from top to bottom.

- ROWS specifies the window in physical units (rows).
- RANGE specifies the window as a logical offset.

You cannot specify this clause unless you have specified the *order_by_clause*.

> **Note:** The value returned by an analytic function with a logical offset is always deterministic. However, the value returned by an analytic function with a physical offset may produce nondeterministic results unless the ordering expression results in a unique ordering. You may have to specify multiple columns in the *order_by_clause* to achieve this unique ordering.

BETWEEN ... AND Use the BETWEEN ... AND clause to specify a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point.

If you omit BETWEEN and specify only one end point, then Oracle considers it the start point, and the end point defaults to the current row.

UNBOUNDED PRECEDING Specify UNBOUNDED PRECEDING to indicate that the window starts at the first row of the partition. This is the start point specification and cannot be used as an end point specification.

UNBOUNDED FOLLOWING Specify UNBOUNDED FOLLOWING to indicate that the window ends at the last row of the partition. This is the end point specification and cannot be used as a start point specification.

CURRENT ROW As a start point, CURRENT ROW specifies that the window begins at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the end point cannot be *value_expr* PRECEDING.

As an end point, CURRENT ROW specifies that the window ends at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the start point cannot be *value_expr* FOLLOWING.

***value_expr* PRECEDING or *value_expr* FOLLOWING** For RANGE or ROW:

- If *value_expr* FOLLOWING is the start point, then the end point must be *value_expr* FOLLOWING.
- If *value_expr* PRECEDING is the end point, then the start point must be *value_expr* PRECEDING.

If you are defining a logical window defined by an interval of time in numeric format, then you may need to use conversion functions.

> **See Also:** NUMTOYMINTERVAL on page 6-109 and NUMTODSINTERVAL on page 6-108 for information on converting numeric times into intervals

If you specified ROWS:

- *value_expr* is a physical offset. It must be a constant or expression and must evaluate to a positive numeric value.
- If *value_expr* is part of the start point, then it must evaluate to a row before the end point.

If you specified RANGE:

- *value_expr* is a logical offset. It must be a constant or expression that evaluates to a positive numeric value or an interval literal.

  > **See Also:** "Literals" on page 2-54 for information on interval literals

- You can specify only one expression in the *order_by_clause*
- If *value_expr* evaluates to a numeric value, then the ORDER BY *expr* must be a NUMBER or DATE datatype.
- If *value_expr* evaluates to an interval value, then the ORDER BY *expr* must be a DATE datatype.

If you omit the *windowing_clause* entirely, then the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Analytic functions are commonly used in data warehousing environments. The analytic functions follow. Functions followed by an asterisk (*) allow the full syntax, including the *windowing_clause*.

AVG *
CORR *
COVAR_POP *
COVAR_SAMP *
COUNT *
CUME_DIST
DENSE_RANK
FIRST
FIRST_VALUE *
LAG
LAST
LAST_VALUE *
LEAD
MAX *
MIN *
NTILE
PERCENT_RANK
PERCENTILE_CONT
PERCENTILE_DISC
RANK
RATIO_TO_REPORT
REGR_ (Linear Regression) Functions *
ROW_NUMBER
STDDEV *
STDDEV_POP *
STDDEV_SAMP *
SUM *
VAR_POP *
VAR_SAMP *
VARIANCE *

See Also: *Oracle9i Data Warehousing Guide* for more information on these functions, and for scenarios illustrating their use

Object Reference Functions

Object reference functions manipulate REFs, which are references to objects of specified object types. The object reference functions are:

DEREF
MAKE_REF
REF
REFTOHEX
VALUE

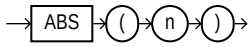
See Also: *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for more information about REFs

Alphabetical Listing of SQL Functions

ABS

Syntax

abs::=



Purpose

ABS returns the absolute value of *n*.

Examples

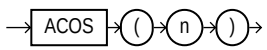
The following example returns the absolute value of -15:

```
SELECT ABS(-15) "Absolute" FROM DUAL;

  Absolute
----------
        15
```

ACOS

Syntax

acos::=



Purpose

ACOS returns the arc cosine of *n*. The argument *n* must be in the range of -1 to 1, and the function returns values in the range of 0 to π, expressed in radians.

Examples

The following example returns the arc cosine of .3:

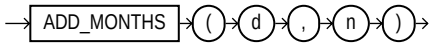
```
SELECT ACOS(.3)"Arc_Cosine" FROM DUAL;

Arc_Cosine
```

```
----------
1.26610367
```

ADD_MONTHS

Syntax

add_months::=



Purpose

ADD_MONTHS returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Examples

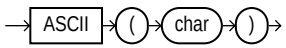
The following example returns the month after the *hire_date* in the sample table employees:

```
SELECT TO_CHAR(
     ADD_MONTHS(hire_date,1),
     'DD-MON-YYYY') "Next month"
     FROM employees
     WHERE last_name = 'Baer';

Next Month
-----------
07-JUL-1994
```

ASCII

Syntax

ascii::=



Purpose

ASCII returns the decimal representation in the database character set of the first character of *char*.

char can be of datatype CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The value returned is of datatype NUMBER. If your database character set is 7-bit ASCII, then this function returns an ASCII value. If your database character set is EBCDIC Code, then this function returns an EBCDIC value. There is no corresponding EBCDIC character function.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

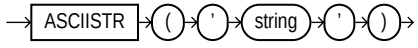
The following example returns the ASCII decimal equivalent of the letter Q:

```
SELECT ASCII('Q') FROM DUAL;

ASCII('Q')
----------
        81
```

ASCIISTR

Syntax

asciistr::=



Purpose

ASCIISTR takes as its argument a string in any character set and returns an ASCII string in the database character set. The value returned contains only characters that appear in SQL, plus the forward slash (/). Non-ASCII characters are converted to their Unicode (UTF-16) binary code value.

See Also: *Oracle9i Database Globalization Support Guide* for information on Unicode character sets and character semantics

Examples

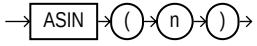
The following example returns the ASCII string equivalent of the text string "ABÄCDE":

```
SELECT ASCIISTR('ABÄCDE') FROM DUAL;

ASCIISTR('FLAUW
---------------
AB\00C4CDE
```

ASIN

Syntax

asin::=



Purpose

ASIN returns the arc sine of *n*. The argument *n* must be in the range of -1 to 1, and the function returns values in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Examples

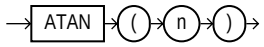
The following example returns the arc sine of .3:

```
SELECT ASIN(.3) "Arc_Sine" FROM DUAL;

 Arc_Sine
----------
.304692654
```

ATAN

Syntax

atan::=



Purpose

ATAN returns the arc tangent of *n*. The argument *n* can be in an unbounded range, and the function returns values in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Examples

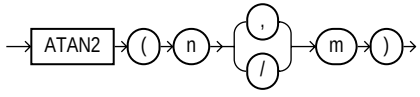
The following example returns the arc tangent of .3:

```
SELECT ATAN(.3) "Arc_Tangent" FROM DUAL;

Arc_Tangent
-----------
.291456794
```

ATAN2

Syntax

atan2::=



Purpose

ATAN2 returns the arc tangent of *n* and *m*. The argument *n* can be in an unbounded range, and the function returns values in the range of $-\pi$ to π, depending on the signs of *n* and *m*, and are expressed in radians. ATAN2(*n*, *m*) is the same as ATAN2(*n*/*m*)

Examples

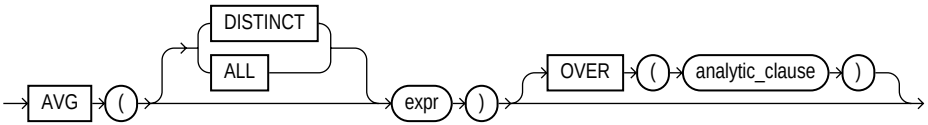
The following example returns the arc tangent of .3 and .2:

```
SELECT ATAN2(.3, .2) "Arc_Tangent2" FROM DUAL;

Arc_Tangent2
------------
  .982793723
```

AVG

Syntax

avg::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

AVG returns average value of *expr*. You can use it as an aggregate or analytic function.

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the average salary of all employees in the hr.employees table:

```
SELECT AVG(salary) "Average" FROM employees;
```

```
 Average
--------
    6425
```

Analytic Example

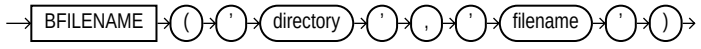
The following example calculates, for each employee in the `employees` table, the average salary of the employees reporting to the same manager who were hired in the range just before through just after the employee:

```
SELECT manager_id, last_name, hire_date, salary,
   AVG(salary) OVER (PARTITION BY manager_id ORDER BY hire_date
   ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS c_mavg
   FROM employees;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     C_MAVG
---------- ------------------------- --------- ---------- ----------
       100 Kochhar                   21-SEP-89      17000      17000
       100 De Haan                   13-JAN-93      17000      15000
       100 Raphaely                  07-DEC-94      11000 11966.6667
       100 Kaufling                  01-MAY-95       7900 10633.3333
       100 Hartstein                 17-FEB-96      13000 9633.33333
       100 Weiss                     18-JUL-96       8000 11666.6667
       100 Russell                   01-OCT-96      14000 11833.3333
.
.
.
```

BFILENAME

Syntax

bfilename::=



Purpose

`BFILENAME` returns a `BFILE` locator that is associated with a physical LOB binary file on the server's file system.

- '*directory*' is a database object that serves as an alias for a full path name on the server's file system where the files are actually located
- '*filename*' is the name of the file in the server's file system

You must create the directory object and associate a BFILE value with a physical file before you can use them as arguments to BFILENAME in a SQL, PL/SQL, DBMS_LOB package, or OCI operation.

See Also:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* and *Oracle Call Interface Programmer's Guide* for more information on LOBs
- CREATE DIRECTORY on page 13-46

Examples

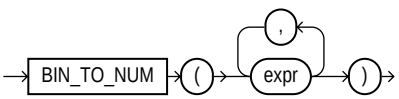
The following example inserts a row into the sample table pm.print_media. The example uses the BFILENAME function to identify a binary file on the server's file system:

```
CREATE DIRECTORY media_dir AS '/demo/schema/product_media';

INSERT INTO print_media (product_id, ad_id, ad_graphic)
   VALUES (3000, 31001,
      bfilename('media_dir', 'modem_comp_ad.gif'));
```

BIN_TO_NUM

Syntax

bin_to_num::=



Purpose

BIN_TO_NUM converts a bit vector to its equivalent number. Each argument to this function represents a bit in the bit vector. Each *expr* must evaluate to 0 or 1. This function returns Oracle NUMBER.

BIN_TO_NUM is useful in data warehousing applications for selecting groups of interest from a materialized view using grouping sets.

See Also:

- *group_by_clause* on page 18-21 for information on GROUPING SETS syntax
- *Oracle9i Data Warehousing Guide* for information on data aggregation in general

Examples

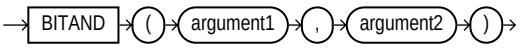
The following example converts a binary value to a number:

```
SELECT BIN_TO_NUM(1,0,1,0) FROM DUAL;

BIN_TO_NUM(1,0,1,0)
-------------------
                 10
```

BITAND

Syntax

bitand::=



Purpose

BITAND computes an AND operation on the bits of *argument1* and *argument2*, both of which must resolve to nonnegative integers, and returns an integer. This function is commonly used with the DECODE function, as illustrated in the example that follows.

Note: This function does not determine the datatype of the value returned. Therefore, in SQL*Plus, you must specify BITAND in a wrapper, such as TO_NUMBER, which returns a datatype.

Examples

The following represents each order_status in the sample table oe.orders by individual bits. (The example specifies options that can total only 7, so rows with order_status greater than 7 are eliminated.)

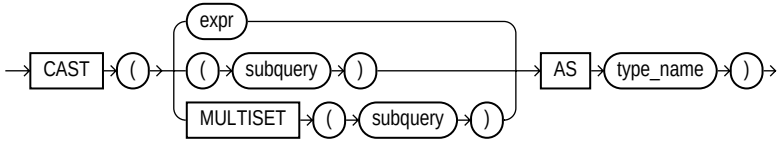
```
SELECT order_id, customer_id,
```

```
  DECODE(BITAND(order_status, 1), 1, 'Warehouse', 'PostOffice')
      Location,
  DECODE(BITAND(order_status, 2), 2, 'Ground', 'Air') Method,
  DECODE(BITAND(order_status, 4), 4, 'Insured', 'Certified') Receipt
  FROM orders
  WHERE order_status < 8;

  ORDER_ID CUSTOMER_ID LOCATION   MET RECEIPT
---------- ----------- ---------- --- ---------
      2458         101 Postoffice Air Certified
      2397         102 Warehouse  Air Certified
      2454         103 Warehouse  Air Certified
      2354         104 Postoffice Air Certified
      2358         105 Postoffice G   Certified
      2381         106 Warehouse  G   Certified
      2440         107 Warehouse  G   Certified
      2357         108 Warehouse  Air Insured
      2394         109 Warehouse  Air Insured
      2435         144 Postoffice G   Insured
      2455         145 Warehouse  G   Insured
.
.
.
```

CAST

Syntax

cast::=



Purpose

CAST converts one built-in datatype or collection-typed value into another built-in datatype or collection-typed value.

CAST lets you convert built-in datatypes or collection-typed values of one type into another built-in datatype or collection type. You can cast an unnamed operand (such as a date or the result set of a subquery) or a named collection (such as a varray or a nested table) into a type-compatible datatype or named collection. The

type_name must be the name of a built-in datatype or collection type and the operand must be a built-in datatype or must evaluate to a collection value.

For the operand, *expr* can be either a built-in datatype or a collection type, and *subquery* must return a single value of collection type or built-in type. MULTISET informs Oracle to take the result set of the subquery and return a collection value. Table 6–1 shows which built-in datatypes can be cast into which other built-in datatypes. (CAST does not support LONG, LONG RAW, any of the LOB datatypes, or the Oracle-supplied types.)

Table 6–1 Casting Built-In Datatypes

	from CHAR, VARCHAR2	from NUMBER	from DATETIME / INTERVAL[b]	from RAW	from ROWID, UROWID	from NCHAR, NVARCHAR2
to CHAR, VARCHAR2	X	X	X	X	X	–
to NUMBER	X	X	–	–	–	–
to DATE, TIMESTAMP, INTERVAL	X	–	X	–	–	–
to RAW	X	–	–	X	–	–
to ROWID, UROWID	X	–	–	–	X[a]	–
to NCHAR, NVARCHAR2	–	X	X	X	X	X

[a] You cannot cast a UROWID to a ROWID if the UROWID contains the value of a ROWID of an index-organized table.

[b] Datetime / Interval includes DATE, TIMESTAMP, TIMESTAMP WITH TIMEZONE, INTERVAL DAY TO SECOND, and INTERVAL YEAR TO MONTH,

If you want to cast a named collection type into another named collection type, then the elements of both collections must be of the same type.

If the result set of *subquery* can evaluate to multiple rows, then you must specify the MULTISET keyword. The rows resulting from the subquery form the elements of the collection value into which they are cast. Without the MULTISET keyword, the subquery is treated as a scalar subquery.

Built-In Datatype Examples

The following examples use the CAST function with scalar datatypes:

```
SELECT CAST('22-OCT-1997' AS TIMESTAMP WITH LOCAL TIME ZONE)
   FROM dual;

SELECT product_id,
   CAST(ad_sourcetext AS VARCHAR2(30))
   FROM print_media;
```

Collection Examples

The CAST examples that follow build on the cust_address_typ found in the sample order entry schema, oe.

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;
/
CREATE TYPE address_array_t AS VARRAY(3) OF cust_address_typ;
/
CREATE TABLE cust_address (
   custno            NUMBER,
   street_address    VARCHAR2(40),
   postal_code       VARCHAR2(10),
   city              VARCHAR2(30),
   state_province    VARCHAR2(10),
   country_id        CHAR(2));

CREATE TABLE cust_short (custno NUMBER, name VARCHAR2(31));

CREATE TABLE states (state_id NUMBER, addresses address_array_t);
```

This example casts a subquery:

```
SELECT s.custno, s.name,
   CAST(MULTISET(SELECT ca.street_address,
                        ca.postal_code,
                        ca.city,
                        ca.state_province,
                        ca.country_id
                 FROM cust_address ca
                 WHERE s.custno = ca.custno)
   AS address_book_t)
FROM cust_short s;
```

CAST converts a varray type column into a nested table:

```
SELECT CAST(s.addresses AS address_book_t)
   FROM states s
   WHERE s.state_id = 111;
```

The following objects create the basis of the example that follows:

```
CREATE TABLE projects
   (employee_id NUMBER, project_name VARCHAR2(10));

CREATE TABLE emps_short
   (employee_id NUMBER, last_name VARCHAR2(10));

CREATE TYPE project_table_typ AS TABLE OF VARCHAR2(10);
   /
```

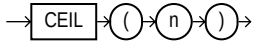
The following example of a MULTISET expression uses these objects:

```
SELECT e.last_name,
   CAST(MULTISET(SELECT p.project_name
   FROM projects p
   WHERE p.employee_id = e.employee_id
   ORDER BY p.project_name)
   AS project_table_typ)
FROM emps_short e;
```

CEIL

Syntax

ceil::=



Purpose

CEIL returns smallest integer greater than or equal to *n*.

Examples

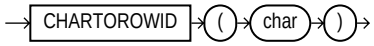
The following example returns the smallest integer greater than or equal to 15.7:

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;

   Ceiling
----------
        16
```

CHARTOROWID

Syntax

chartorowid::=



Purpose

CHARTOROWID converts a value from CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to ROWID datatype.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

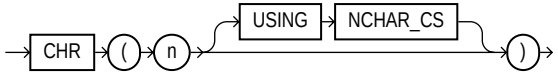
The following example converts a character rowid representation to a rowid. (The function will return a different rowid on different databases).

```
SELECT last_name FROM employees
   WHERE ROWID = CHARTOROWID('AAAFd1AAFAAAABSAA/');

LAST_NAME
-------------------------
Greene
```

CHR

Syntax

chr::=



Purpose

CHR returns the character having the binary equivalent to *n* in either the database character set or the national character set.

If USING NCHAR_CS is not specified, then this function returns the character having the binary equivalent to *n* as a VARCHAR2 value in the database character set.

If USING NCHAR_CS is specified, then this function returns the character having the binary equivalent to *n* as a NVARCHAR2 value in the national character set.

For single-byte character sets, if n > 256, then Oracle returns the binary equivalent of n mod 256. For multibyte character sets, n must resolve to one entire codepoint. Invalid codepoints are not validated, and the result of specifying invalid codepoints is indeterminate.

Note: Use of the CHR function (either with or without the optional USING NCHAR_CS clause) results in code that is not portable between ASCII- and EBCDIC-based machine architectures.

See Also: NCHR on page 6-97

Examples

The following example is run on an ASCII-based machine with the database character set defined as WE8ISO8859P1:

```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog" FROM DUAL;

Dog
---
CAT
```

To produce the same results on an EBCDIC-based machine with the WE8EBCDIC1047 character set, the preceding example would have to be modified as follows:

```
SELECT CHR(195)||CHR(193)||CHR(227) "Dog"
   FROM DUAL;

Dog
---
CAT
```

For multibyte character sets, this sort of concatenation gives different results. For example, given a multibyte character whose hexadecimal value is a1a2 (a1 representing the first byte and a2 the second byte), you must specify for n the decimal equivalent of 'a1a2', or 41378. That is, you must specify:

```
SELECT CHR(41378) FROM DUAL;
```

You cannot specify the decimal equivalent of a1 concatenated with the decimal equivalent of a2, as in the following example:

```
SELECT CHR(161)||CHR(162) FROM DUAL;
```

However, you can concatenate whole multibyte codepoints, as in the following example, which concatenates the multibyte characters whose hexadecimal values are a1a2 and a1a3:

```
SELECT CHR(41378)||CHR(41379) FROM DUAL;
```

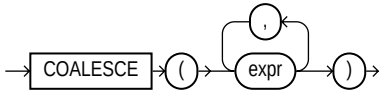
The following example uses the UTF8 character set:

```
SELECT CHR (50052 USING NCHAR_CS) FROM DUAL;

CH
--
Ä
```

COALESCE

Syntax

coalesce::=



Purpose

COALESCE returns the first non-null *expr* in the expression list. At least one expr must not be the literal NULL. If all occurrences of *expr* evaluate to null, then the function returns null.

This function is a generalization of the NVL function.

You can also use COALESCE as a variety of the CASE expression. For example,

```
COALESCE (expr1, expr2)
```

is equivalent to:

```
CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END
```

Similarly,

```
COALESCE (expr1, expr2, ..., exprn), for n>=3
```

is equivalent to:

```
CASE WHEN expr1 IS NOT NULL THEN expr1
   ELSE COALESCE (expr2, ..., exprn) END
```

> **See Also:** NVL on page 6-110 and "CASE Expressions" on page 4-6

Examples

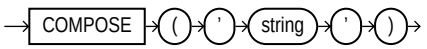
The following example uses the sample `oe.product_information` table to organize a "clearance sale" of products. It gives a 10% discount to all products with a list price. If there is no list price, then the sale price is the minimum price. If there is no minimum price, then the sale price is "5":

```
SELECT product_id, list_price, min_price,
   COALESCE(0.9*list_price, min_price, 5) "Sale"
   FROM product_information
   WHERE supplier_id = 102050;

PRODUCT_ID LIST_PRICE  MIN_PRICE       Sale
---------- ---------- ---------- ----------
      2382        850        731        765
      3355                                5
      1770                    73         73
      2378        305        247      274.5
      1769         48                  43.2
```

COMPOSE

Syntax

compose::=



Purpose

COMPOSE takes as its argument a string in any datatype, and returns a Unicode string in its fully normalized form in the same character set as the input. *string* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. For example, an "o" codepoint qualified by an umlaut codepoint will be returned as the o-umlaut codepoint.

> **See Also:** *Oracle9i Database Concepts* for information on Unicode character sets and character semantics

Examples

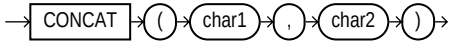
The following example returns the o-umlaut codepoint:

```
SELECT COMPOSE ( 'o' || UNISTR('\0308') ) FROM DUAL;

CO
--
ö
```

CONCAT

Syntax

concat::=



Purpose

CONCAT returns *char1* concatenated with *char2*. Both *char1* and *char2* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is in the same character set as *char1*. Its datatype depends on the datatypes of the arguments.

In concatenations of two different datatypes, Oracle returns the datatype that results in a lossless conversion. Therefore, if one of the arguments is a LOB, then the returned value is a LOB. If one of the arguments is a national datatype, then the returned value is a national datatype. For example:

- CONCAT(CLOB, NCLOB) returns NCLOB
- CONCAT(NCLOB, NCHAR) returns NCLOB

- CONCAT(NCLOB, CHAR) returns NCLOB
- CONCAT(NCHAR, CLOB) returns NCLOB

This function is equivalent to the concatenation operator (||).

> **See Also:** "Concatenation Operator" on page 3-4 for information on the CONCAT operator

Examples

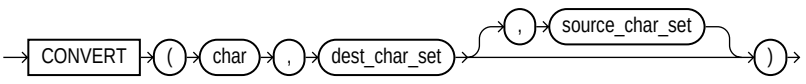
This example uses nesting to concatenate three character strings:

```
SELECT CONCAT(CONCAT(last_name, '''s job category is '),
      job_id) "Job"
   FROM employees
   WHERE employee_id = 152;

Job
---------------------------------------------------
Hall's job category is SA_REP
```

CONVERT

Syntax

convert::=



Purpose

CONVERT converts a character string from one character set to another. The datatype of the returned value is VARCHAR2.

- The *char* argument is the value to be converted. It can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- The *dest_char_set* argument is the name of the character set to which *char* is converted.
- The *source_char_set* argument is the name of the character set in which *char* is stored in the database. The default value is the database character set.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set.

For complete correspondence in character conversion, it is essential that the destination character set contains a representation of all the characters defined in the source character set. Where a character does not exist in the destination character set, a replacement character appears. Replacement characters can be defined as part of a character set definition.

Examples

The following example illustrates character set conversion by converting a Latin-1 string to ASCII. The result is the same as importing the same string from a WE8ISO8859P1 database to a US7ASCII database.

```
SELECT CONVERT('Ä Ê Í Õ Ø A B C D E ', 'US7ASCII', 'WE8ISO8859P1')
   FROM DUAL;

CONVERT('ÄÊÍÕØABCDE'
--------------------
A E I ? ? A B C D E ?
```

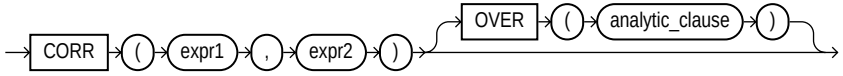
Common character sets include:

- US7ASCII: US 7-bit ASCII character set
- WE8DEC: West European 8-bit character set
- WE8HP: HP West European Laserjet 8-bit character set
- F7DEC: DEC French 7-bit character set
- WE8EBCDIC500: IBM West European EBCDIC Code Page 500
- WE8PC850: IBM PC Code Page 850
- WE8ISO8859P1: ISO 8859-1 West European 8-bit character set

CORR

Syntax

corr::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

CORR returns the coefficient of correlation of a set of number pairs. You can use it as an aggregate or analytic function.

Both *expr1* and *expr2* are number expressions. Oracle applies the function to the set of (*expr1*, *expr2*) after eliminating the pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
COVAR_POP(expr1, expr2) / (STDDEV_POP(expr1) * STDDEV_POP(expr2))
```

The function returns a value of type NUMBER. If the function is applied to an empty set, then it returns null.

> **See Also:**
>
> - "Aggregate Functions" on page 6-7
> - "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the coefficient of correlation between the list prices and minimum prices of products by weight class in the sample view oe.products:

```
SELECT weight_class, CORR(list_price, min_price)
   FROM product_information
   GROUP BY weight_class;

WEIGHT_CLASS CORR(LIST_PRICE,MIN_PRICE)
------------ --------------------------
           1                  .99914795
           2                 .999022941
           3                 .998484472
           4                 .999359909
           5                 .999536087
```

Analytic Example

The following example returns the cumulative coefficient of correlation of monthly sales revenues and monthly units sold from the sample tables sh.sales and sh.times for year 1998:

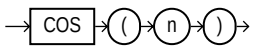
```
SELECT t.calendar_month_number,
   CORR (SUM(s.amount_sold), SUM(s.quantity_sold))
   OVER (ORDER BY t.calendar_month_number) as CUM_CORR
   FROM sales s, times t
      WHERE s.time_id = t.time_id AND calendar_year = 1998
      GROUP BY t.calendar_month_number
      ORDER BY t.calendar_month_number;

CALENDAR_MONTH_NUMBER   CUM_CORR
--------------------- ----------
                    1
                    2          1
                    3 .994309382
                    4 .852040875
                    5 .846652204
                    6 .871250628
                    7 .910029803
                    8 .917556399
                    9 .920154356
                   10  .86720251
                   11 .844864765
                   12 .903542662
```

Correlation functions require more than one row on which to operate, so the first row in the preceding example has no value calculated for it.

COS

Syntax

cos::=



Purpose

COS returns the cosine of *n* (an angle expressed in radians).

Examples

The following example returns the cosine of 180 degrees:

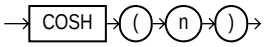
```
SELECT COS(180 * 3.14159265359/180)
   "Cosine of 180 degrees" FROM DUAL;
```

```
Cosine of 180 degrees
---------------------
                   -1
```

COSH

Syntax

cosh::=



Purpose

COSH returns the hyperbolic cosine of *n*.

Examples

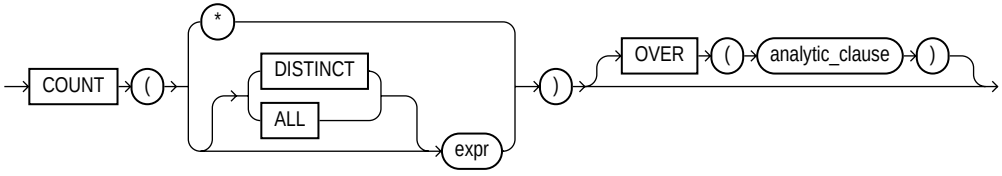
The following example returns the hyperbolic cosine of zero:

```
SELECT COSH(0) "Hyperbolic cosine of 0" FROM DUAL;

Hyperbolic cosine of 0
----------------------
                     1
```

COUNT

Syntax

count::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

COUNT returns the number of rows in the query. You can use it as an aggregate or analytic function.

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

If you specify *expr*, then COUNT returns the number of rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.

If you specify the asterisk (*), then this function returns all rows, including duplicates and nulls. COUNT never returns null.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Examples

The following examples use COUNT as an aggregate function:

```
SELECT COUNT(*) "Total" FROM employees;

     Total
----------
       107

SELECT COUNT(*) "Allstars" FROM employees
   WHERE commission_pct > 0;

 Allstars
---------
       35

SELECT COUNT(commission_pct) "Count" FROM employees;

     Count
----------
        35

SELECT COUNT(DISTINCT manager_id) "Managers" FROM employees;
```

```
  Managers
----------
        18
```

Analytic Example

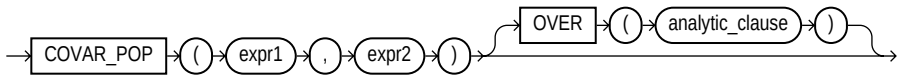
The following example calculates, for each employee in the `employees` table, the moving count of employees earning salaries in the range $50 less than through $150 greater than the employee's salary.

```
SELECT last_name, salary,
   COUNT(*) OVER (ORDER BY salary RANGE BETWEEN 50 PRECEDING
      AND 150 FOLLOWING) AS mov_count FROM employees;

LAST_NAME                      SALARY  MOV_COUNT
-------------------------- ---------- ----------
Olson                            2100          3
Markle                           2200          2
Philtanker                       2200          2
Landry                           2400          8
Gee                              2400          8
Colmenares                       2500         10
Patel                            2500         10
.
.
.
```

COVAR_POP

Syntax

covar_pop::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

COVAR_POP returns the population covariance of a set of number pairs. You can use it as an aggregate or analytic function.

Both *expr1* and *expr2* are number expressions. Oracle applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr2) * SUM(expr1) / n) / n
```

where *n* is the number of (*expr1*, *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type NUMBER. If the function is applied to an empty set, then it returns null.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the population covariance for the sales revenue amount and the units sold for each year from the sample table sh.sales:

```
SELECT t.calendar_month_number,
   COVAR_POP(s.amount_sold, s.quantity_sold) AS covar_pop,
   COVAR_SAMP(s.amount_sold, s.quantity_sold) AS covar_samp
   FROM sales s, times t
   WHERE s.time_id = t.time_id
   AND t.calendar_year = 1998
   GROUP BY t.calendar_month_number;

CALENDAR_MONTH_NUMBER  COVAR_POP COVAR_SAMP
--------------------- ---------- ----------
                    1 5437.68586 5437.88704
                    2 5923.72544 5923.99139
                    3 6040.11777 6040.38623
                    4 5946.67897 5946.92754
                    5 5986.22483  5986.4463
                    6 5726.79371 5727.05703
                    7 5491.65269  5491.9239
                    8 5672.40362 5672.66882
                    9 5741.53626 5741.80025
                   10  5050.5683 5050.78195
                   11 5256.50553 5256.69145
                   12  5411.2053 5411.37709
```

Analytic Example

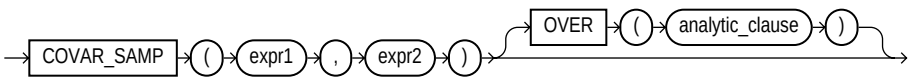
The following example calculates cumulative sample covariance of the list price and minimum price of the products in the sample schema oe:

```
SELECT product_id, supplier_id,
   COVAR_POP(list_price, min_price)
      OVER (ORDER BY product_id, supplier_id)
         AS CUM_COVP,
   COVAR_SAMP(list_price, min_price)
      OVER (ORDER BY product_id, supplier_id)
        AS CUM_COVS
   FROM product_information p
   WHERE category_id = 29
   ORDER BY product_id, supplier_id;

PRODUCT_ID SUPPLIER_ID   CUM_COVP   CUM_COVS
---------- ----------- ---------- ----------
      1774      103088          0
      1775      103087    1473.25     2946.5
      1794      103096 1702.77778 2554.16667
      1825      103093    1926.25 2568.33333
      2004      103086     1591.4    1989.25
      2005      103086     1512.5       1815
      2416      103088 1475.97959 1721.97619
.
.
.
```

COVAR_SAMP

Syntax

covar_samp::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

COVAR_SAMP returns the sample covariance of a set of number pairs. You can use it as an aggregate or analytic function.

Both *expr1* and *expr2* are number expressions. Oracle applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Then Oracle makes the following computation:

```
(SUM(expr1 * expr2) - SUM(expr1) * SUM(expr2) / n) / (n-1)
```

where *n* is the number of (*expr1*, *expr2*) pairs where neither *expr1* nor *expr2* is null.

The function returns a value of type NUMBER. If the function is applied to an empty set, then it returns null.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the population covariance for the sales revenue amount and the units sold for each year from the sample table sh.sales:

```
SELECT t.calendar_month_number,
   COVAR_POP(s.amount_sold, s.quantity_sold) AS covar_pop,
   COVAR_SAMP(s.amount_sold, s.quantity_sold) AS covar_samp
   FROM sales s, times t
   WHERE s.time_id = t.time_id
   AND t.calendar_year = 1998
   GROUP BY t.calendar_month_number;

CALENDAR_MONTH_NUMBER  COVAR_POP COVAR_SAMP
--------------------- ---------- ----------
                    1 5437.68586 5437.88704
                    2 5923.72544 5923.99139
                    3 6040.11777 6040.38623
                    4 5946.67897 5946.92754
                    5 5986.22483  5986.4463
                    6 5726.79371 5727.05703
                    7 5491.65269  5491.9239
                    8 5672.40362 5672.66882
                    9 5741.53626 5741.80025
                   10  5050.5683 5050.78195
                   11 5256.50553 5256.69145
                   12  5411.2053 5411.37709
```

Analytic Example

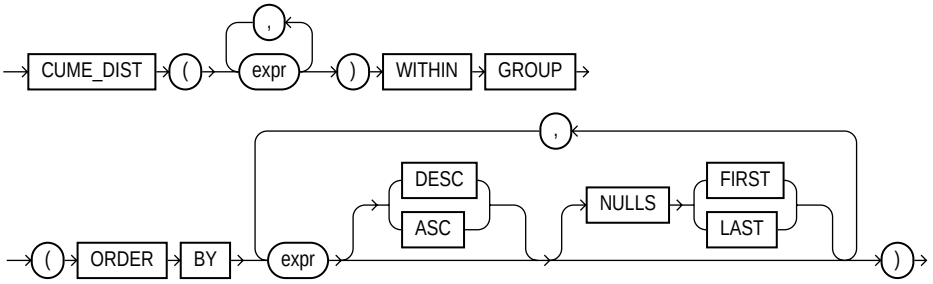
The following example calculates cumulative sample covariance of the list price and minimum price of the products in the sample schema oe:

```
SELECT product_id, supplier_id,
   COVAR_POP(list_price, min_price)
      OVER (ORDER BY product_id, supplier_id)
         AS CUM_COVP,
   COVAR_SAMP(list_price, min_price)
      OVER (ORDER BY product_id, supplier_id)
        AS CUM_COVS
   FROM product_information p
   WHERE category_id = 29
   ORDER BY product_id, supplier_id;

PRODUCT_ID SUPPLIER_ID   CUM_COVP   CUM_COVS
---------- ----------- ---------- ----------
      1774      103088          0
      1775      103087    1473.25     2946.5
      1794      103096 1702.77778 2554.16667
      1825      103093    1926.25 2568.33333
      2004      103086     1591.4    1989.25
      2005      103086     1512.5       1815
      2416      103088 1475.97959 1721.97619
.
.
.
```

CUME_DIST

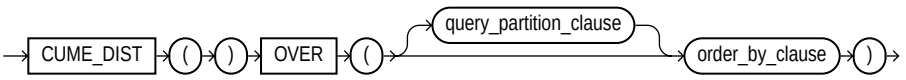
Aggregate Syntax

cume_dist_aggregate::=



Analytic Syntax

cume_dist_analytic::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

`CUME_DIST` calculates the cumulative distribution of a value in a group of values. The range of values returned by `CUME_DIST` is >0 to <=1. Tie values always evaluate to the same cumulative distribution value.

- As an aggregate function, `CUME_DIST` calculates, for a hypothetical row R identified by the arguments of the function and a corresponding sort specification, the relative position of row R among the rows in the aggregation group. Oracle makes this calculation as if the hypothetical row R were inserted into the group of rows to be aggregated over. The arguments of the function identify a single hypothetical row within each aggregate group. Therefore, they must all evaluate to constant expressions within each aggregate group. The constant argument expressions and the expressions in the `ORDER BY` clause of the aggregate match by position. Therefore, the number of arguments must be the same and their types must be compatible.

- As an analytic function, CUME_DIST computes the relative position of a specified value in a group of values. For a row R, assuming ascending ordering, the CUME_DIST of R is the number of rows with values lower than or equal to the value of R, divided by the number of rows being evaluated (the entire query result set or a partition).

Aggregate Example

The following example calculates the cumulative distribution of a hypothetical employee with a salary of $15,500 and commission rate of 5% among the employees in the sample table oe.employees:

```
SELECT CUME_DIST(15500, .05) WITHIN GROUP
   (ORDER BY salary, commission_pct) "Cume-Dist of 15500"
   FROM employees;

Cume-Dist of 15500
------------------
        .972222222
```

Analytic Example

The following example calculates the salary percentile for each employee in the purchasing area. For example, 40% of clerks have salaries less than or equal to Himuro.

```
SELECT job_id, last_name, salary, CUME_DIST()
   OVER (PARTITION BY job_id ORDER BY salary) AS cume_dist
   FROM employees
   WHERE job_id LIKE 'PU%';

JOB_ID     LAST_NAME                     SALARY  CUME_DIST
---------- ------------------------- ---------- ----------
PU_CLERK   Colmenares                      2500         .2
PU_CLERK   Himuro                          2600         .4
PU_CLERK   Tobias                          2800         .6
PU_CLERK   Baida                           2900         .8
PU_CLERK   Khoo                            3100          1
PU_MAN     Raphaely                       11000          1
```

CURRENT_DATE

Syntax

current_date::=

Purpose

CURRENT_DATE returns the current date in the session time zone, in a value in the Gregorian calendar of datatype DATE.

Examples

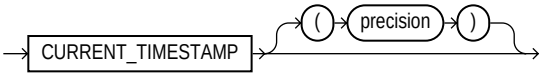
The following example illustrates that CURRENT_DATE is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-05:00          29-MAY-2000 13:14:03

ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-08:00          29-MAY-2000 10:14:33
```

CURRENT_TIMESTAMP

Syntax

current_timestamp::=



Purpose

CURRENT_TIMESTAMP returns the current date and time in the session time zone, in a value of datatype TIMESTAMP WITH TIME ZONE. The time zone displacement reflects the current local time of the SQL session. If you omit precision, then the default is 6. The difference between this function and LOCALTIMESTAMP is that CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value while LOCALTIMESTAMP returns a TIMESTAMP value.

In the optional argument, *precision* specifies the fractional second precision of the time value returned.

Examples

The following example illustrates that CURRENT_TIMESTAMP is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE CURRENT_TIMESTAMP
--------------- ---------------------------------------------------
-05:00          04-APR-00 01.17.56.917550 PM -05:00

ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;

SESSIONTIMEZONE CURRENT_TIMESTAMP
--------------- ---------------------------------------------------
-08:00          04-APR-00 10.18.21.366065 AM -08:00
```

If you use the CURRENT_TIMESTAMP with a format mask, take care that the format mask matches the value returned by the function. For example, consider the following table:

```
CREATE TABLE current_test (col1 TIMESTAMP WITH TIME ZONE);
```

The following statement fails because the mask does not include the TIME ZONE portion of the type returned by the function:

```
INSERT INTO current_test VALUES
  (TO_TIMESTAMP_TZ(CURRENT_TIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM'));
```

The following statement uses the correct format mask to match the return type of CURRENT_TIMESTAMP:

```
INSERT INTO current_test VALUES (TO_TIMESTAMP_TZ
    (CURRENT_TIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM TZH:TZM'));
```

DBTIMEZONE

Syntax

dbtimezone::=

Purpose

DBTIMEZONE returns the value of the database time zone. The return type is a time zone offset (a character type in the format '[+|-]TZH:TZM') or a time zone region name, depending on how the user specified the database time zone value in the most recent CREATE DATABASE or ALTER DATABASE statement.

Examples

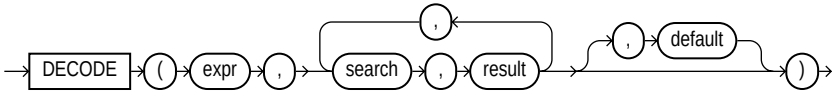
The following example assumes that the database time zone is set to UTC time zone:

```
SELECT DBTIMEZONE FROM DUAL;

DBTIME
------
-08:00
```

DECODE

Syntax

decode::=



Purpose

DECODE compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, then Oracle returns the corresponding *result*. If no match is found, then Oracle returns *default*. If *default* is omitted, then Oracle returns null.

If *expr* and *search* contain character data, then Oracle compares them using nonpadded comparison semantics. *expr*, *search*, and *result* can be any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The string returned is of VARCHAR2 datatype and is in the same character set as the first *result* parameter.

The *search*, *result*, and *default* values can be derived from expressions. Oracle evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, Oracle never evaluates a *search* if a previous *search* is equal to *expr*.

Oracle automatically converts *expr* and each *search* value to the datatype of the first *search* value before comparing. Oracle automatically converts the return value to the same datatype as the first *result*. If the first *result* has the datatype CHAR or if the first *result* is null, then Oracle converts the return value to the datatype VARCHAR2.

In a DECODE function, Oracle considers two nulls to be equivalent. If *expr* is null, then Oracle returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE function, including *expr*, *searches*, *results*, and *default*, is 255.

See Also:

- "Datatype Comparison Rules" on page 2-45 for information on comparison semantics
- "Data Conversion" on page 2-48 for information on datatype conversion in general
- "Implicit and Explicit Data Conversion" on page 2-48 for information on the drawbacks of implicit conversion

Examples

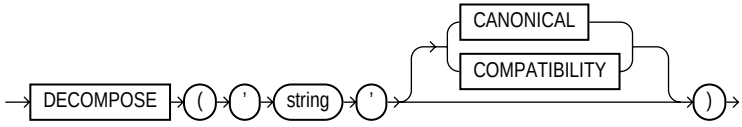
This example decodes the value `warehouse_id`. If `warehouse_id` is 1, then the function returns '`Southlake`'; if `warehouse_id` is 2, then it returns '`San Francisco`'; and so forth. If `warehouse_id` is not 1, 2, 3, or 4, then the function returns '`Non-domestic`'.

```
SELECT product_id,
       DECODE (warehouse_id, 1, 'Southlake',
                             2, 'San Francisco',
                             3, 'New Jersey',
                             4, 'Seattle',
                                'Non-domestic')
       quantity_on_hand FROM inventories;
```

DECOMPOSE

Syntax

decompose::=



Purpose

`DECOMPOSE` is valid only for Unicode characters. `DECOMPOSE` takes as its argument a string in any datatype and returns a Unicode string after decomposition in the same character set as the input. For example, an o-umlaut codepoint will be returned as the "o" codepoint followed by an umlaut codepoint.

- *string* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- CANONICAL causes canonical decomposition, which allows recomposition (for example, with the COMPOSE function) to the original string. This is the default.
- COMPATIBILITY causes decomposition in compatibility mode. In this mode, recomposition is not possible. This mode is useful, for example, when decomposing half-width and full-width *katakana* characters, where recomposition might not be desirable without external formatting or style information.

> **See Also:** *Oracle9i Database Concepts* for information on Unicode character sets and character semantics

Examples

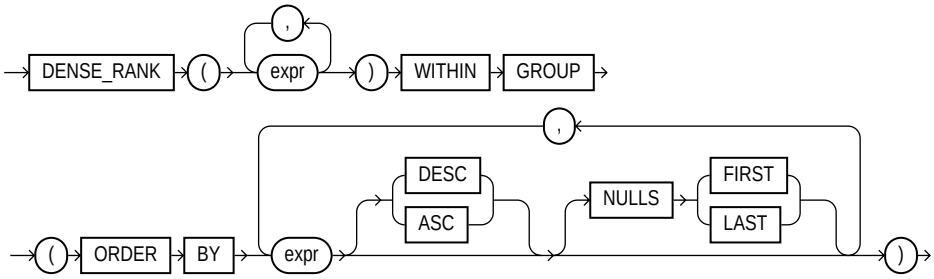
The following example decomposes the string "Châteaux" into its component codepoints:

```
SELECT DECOMPOSE ('Châteaux') FROM DUAL;

DECOMPOSE
---------
Cha^teaux
```

DENSE_RANK

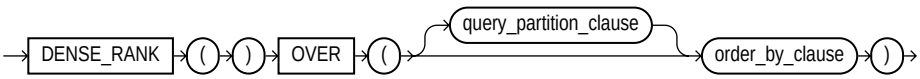
Aggregate Syntax

dense_rank_aggregate::=



Analytic Syntax

dense_rank_analytic::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

DENSE_RANK computes the rank of a row in an ordered group of rows. The ranks are consecutive integers beginning with 1. The largest rank value is the number of unique values returned by the query. Rank values are not skipped in the event of ties. Rows with equal values for the ranking criteria receive the same rank.

- As an aggregate function, DENSE_RANK calculates the dense rank of a hypothetical row identified by the arguments of the function with respect to a given sort specification. The arguments of the function must all evaluate to constant expressions within each aggregate group, because they identify a single row within each group. The constant argument expressions and the expressions in the *order_by_clause* of the aggregate match by position. Therefore, the number of arguments must be the same and types must be compatible.

- As an analytic function, DENSE_RANK computes the rank of each row returned from a query with respect to the other rows, based on the values of the *value_exprs* in the *order_by_clause*.

Aggregate Example

The following example computes the ranking of a hypothetical employee with the salary $15,500 and a commission of 5% in the sample table oe.employees:

```
SELECT DENSE_RANK(15500, .05) WITHIN GROUP
   (ORDER BY salary DESC, commission_pct) "Dense Rank"
   FROM employees;

         Dense Rank
-------------------
                  3
```

Analytic Example

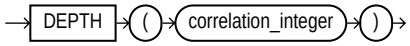
The following statement selects the department name, employee name, and salary of all employees who work in the HUMAN RESOURCES or PURCHASING department, and then computes a rank for each unique salary in each of the two departments. The salaries that are equal receive the same rank. Compare this example with the example for RANK on page 6-120.

```
SELECT d.department_name, e.last_name, e.salary, DENSE_RANK()
   OVER (PARTITION BY e.department_id ORDER BY e.salary) as drank
   FROM employees e, departments d
   WHERE e.department_id = d.department_id
   AND d.department_id IN ('30', '40');

DEPARTMENT_NAME         LAST_NAME                 SALARY      DRANK
----------------------- ------------------ ---------- ----------
Purchasing              Colmenares                2500          1
Purchasing              Himuro                    2600          2
Purchasing              Tobias                    2800          3
Purchasing              Baida                     2900          4
Purchasing              Khoo                      3100          5
Purchasing              Raphaely                 11000          6
Human Resources         Marvis                    6500
```

DEPTH

Syntax

depth::=



Purpose

DEPTH is an ancillary function used only with the UNDER_PATH and EQUALS_PATH conditions. It returns the number of levels in the path specified by the UNDER_PATH condition with the same correlation variable.

The *correlation_integer* can be any integer. Use it to correlate this ancillary function with its primary condition if the statement contains multiple primary conditions. Values less than 1 are treated as 1.

See Also:

- EQUALS_PATH on page 5-13, UNDER_PATH on page 5-20
- the related function PATH on page 6-112

Examples

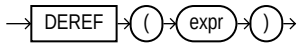
The EQUALS_PATH and UNDER_PATH conditions can take two ancillary functions, one of which is DEPTH. The following example shows the use of both ancillary functions. The example assumes the existence of the XMLSchema warehouses.xsd (created in "Using XML in SQL Statements" on page D-11).

```
SELECT PATH(1), DEPTH(2)
   FROM RESOURCE_VIEW
   WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
     AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;

PATH(1)                          DEPTH(2)
-------------------------------- --------
/www.oracle.com                         1

/www.oracle.com/xwarehouses.xsd         2
```

DEREF

Syntax

deref::=



Purpose

DEREF returns the object reference of argument *expr*, where *expr* must return a REF to an object. If you do not use this function in a query, then Oracle returns the object ID of the REF instead, as shown in the example that follows.

See Also: MAKE_REF on page 6-91

Examples

The sample schema oe contains an object type cust_address_typ (its creation is duplicated in the example that follows). The following example creates a table of cust_address_typ_new, and another table with one column that is a REF to cust_address_typ:

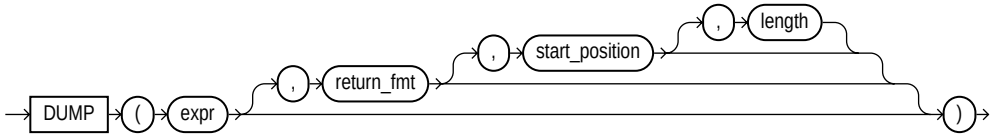
```
CREATE TYPE cust_address_typ_new AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
/
CREATE TABLE address_table OF cust_address_typ_new;

CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ_new
   SCOPE IS address_table);

INSERT INTO address_table VALUES
   ('1 First', 'G45 EU8', 'Paris', 'CA', 'US');

INSERT INTO customer_addresses
  SELECT 999, REF(a) FROM address_table a;

SELECT address FROM customer_addresses;
```

```
ADDRESS
--------------------------------------------------------------------------------
000022020876B2245DBE325C5FE03400400B40DCB176B2245DBE305C5FE03400400B40DCB1

SELECT DEREF(address) FROM customer_addresses;

DEREF(ADDRESS)(STREET_ADDRESS, POSTAL_CODE, CITY, STATE_PROVINCE, COUNTRY_ID)
--------------------------------------------------------------------------------
CUST_ADDRESS_TYP('1 First', 'G45 EU8', 'Paris', 'CA', 'US')
```

DUMP

Syntax

dump::=



Purpose

DUMP returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of *expr*. The returned result is always in the database character set. For the datatype corresponding to each code, see Table 2–1 on page 2-6.

The argument *return_fmt* specifies the format of the return value and can have any of the following values:

- 8 returns result in octal notation.
- 10 returns result in decimal notation.
- 16 returns result in hexadecimal notation.
- 17 returns result as single characters.

By default, the return value contains no character set information. To retrieve the character set name of *expr*, specify any of the preceding format values, plus 1000. For example, a *return_fmt* of 1008 returns the result in octal, plus provides the character set name of *expr*.

The arguments *start_position* and *length* combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

If *expr* is null, then this function returns a null.

> **Note:** This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

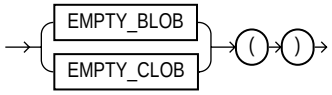
The following examples show how to extract dump information from a string expression and a column:

```
SELECT DUMP('abc', 1016)
   FROM DUAL;

DUMP('ABC',1016)
----------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63

SELECT DUMP(last_name, 8, 3, 2) "OCTAL"
   FROM employees
   WHERE last_name = 'Hunold';

OCTAL
-----------------------------------------------------------------
Typ=1 Len=6: 156,157

SELECT DUMP(last_name, 10, 3, 2) "ASCII"
   FROM employees
   WHERE last_name = 'Hunold';

ASCII
-----------------------------------------------------------------
Typ=1 Len=6: 110,111
```

EMPTY_BLOB, EMPTY_CLOB

Syntax

empty_LOB::=



Purpose

EMPTY_BLOB and EMPTY_CLOB return an empty LOB locator that can be used to initialize a LOB variable or, in an INSERT or UPDATE statement, to initialize a LOB column or attribute to EMPTY. EMPTY means that the LOB is initialized, but not populated with data.

Restriction: You cannot use the locator returned from this function as a parameter to the DBMS_LOB package or the OCI.

Examples

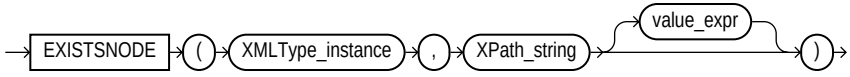
The following example initializes the ad_photo column of the sample pm.print_media table to EMPTY:

```
UPDATE print_media SET ad_photo = EMPTY_BLOB();
```

EXISTSNODE

Syntax

existsnode::=



Purpose

EXISTSNODE determines whether traversal of the document using the path results in any nodes. It takes as arguments the XMLType instance containing an XML document and a VARCHAR2 XPath string designating a path.

The return value is NUMBER:

- 0 if no nodes remain after applying the XPath traversal on the document

- 1 if any nodes remain

Examples

The following example tests for the existence of the `/Warehouse/Dock` node in the XML path of the `warehouse_spec` column of the sample table `oe.warehouses`:

```
SELECT warehouse_id, warehouse_name
   FROM warehouses
   WHERE EXISTSNODE(warehouse_spec, '/Warehouse/Docks') = 1;

WAREHOUSE_ID WAREHOUSE_NAME
------------ ----------------------------------
           1 Southlake, Texas
           2 San Francisco
           4 Seattle, Washington
```

EXP

Syntax

exp::=

Purpose

EXP returns e raised to the *n*th power, where e = 2.71828183 ...

Examples

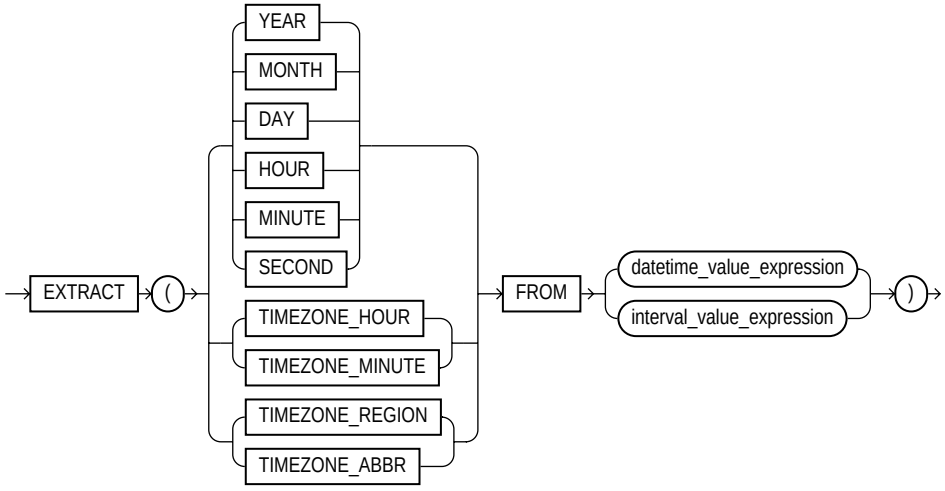
The following example returns e to the 4th power:

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
------------------
          54.59815
```

EXTRACT (datetime)

Syntax

extract_datetime::=



Purpose

EXTRACT extracts and returns the value of a specified datetime field from a datetime or interval value expression. When you extract a TIMEZONE_REGION or TIMEZONE_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC. For a listing of time zone names and their corresponding abbreviations, query the V$TIMEZONE_NAMES dynamic performance view.

Note: The field you are extracting must be a field of the *datetime_value_expr* or *interval_value_expr*. For example, you can extract only YEAR, MONTH, and DAY from a DATE value. Likewise, you can extract TIMEZONE_HOUR and TIMEZONE_MINUTE only from the TIMESTAMP WITH TIME ZONE datatype.

See Also:

- "Datetime / Interval Arithmetic" on page 2-24 for a description of *datetime_value_expr* and *interval_value_expr*
- *Oracle9i Database Reference* for information on the dynamic performance views

Examples

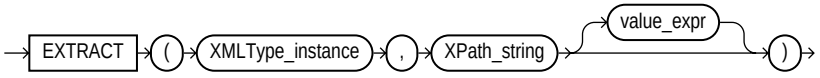
The following example returns the number 1998.

```
SELECT EXTRACT(YEAR FROM DATE '1998-03-07') FROM DUAL;

EXTRACT(YEARFROMDATE'1998-03-07')
---------------------------------
                             1998
```

EXTRACT (XML)

Syntax

extract_xml::=



Purpose

EXTRACT (XML) is similar to the EXISTSNODE function. It applies a VARCHAR2 XPath string and returns an XMLType instance containing an XML fragment.

Examples

The following example extracts the value of the /Warehouse/Dock node of the XML path of the warehouse_spec column in the sample table oe.warehouses:

```
SELECT warehouse_name,
   EXTRACT(warehouse_spec, '/Warehouse/Docks')
   "Number of Docks"
   FROM warehouses
   WHERE warehouse_spec IS NOT NULL;

WAREHOUSE_NAME       Number of Docks
-------------------- --------------------
```

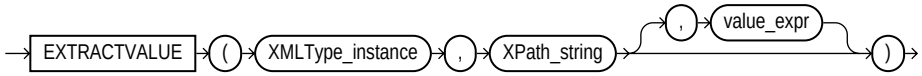
```
Southlake, Texas          <Docks>2</Docks>
San Francisco             <Docks>1</Docks>
New Jersey                <Docks/>
Seattle, Washington       <Docks>3</Docks>
```

Compare this example with the example for EXTRACTVALUE on page 6-63, which returns the scalar value of the XML fragment.

EXTRACTVALUE

Syntax

extractvalue::=



The EXTRACTVALUE function takes as arguments an XMLType instance and an XPath expression and returns a scalar value of the resultant node. The result must be a single node and be either a text node, attribute, or element. If the result is an element, the element must have a single text node as its child, and it is this value that the function returns. If the specified XPath points to a node with more than one child, or if the node pointed to has a non-text node child, Oracle returns an error.

For documents based on XML schemas, if Oracle can infer the type of the return value, then a scalar value of the appropriate type is returned. Otherwise, the result is of type VARCHAR2. For documents that are not based on XML schemas, the return type is always VARCHAR2.

Examples

The following example takes as input the same arguments as the example for EXTRACT (XML) on page 6-62. Instead of returning an XML fragment, as does the EXTRACT function, it returns the scalar value of the XML fragment:

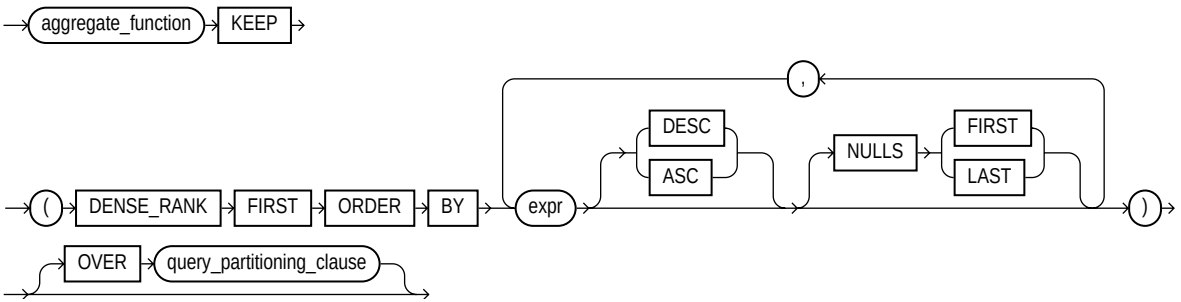
```
SELECT warehouse_name,
   EXTRACTVALUE(e.warehouse_spec, '/Warehouse/Docks')
   "Docks"
   FROM warehouses e
   WHERE warehouse_spec IS NOT NULL;
```

```
WAREHOUSE_NAME       Docks
-------------------- ------------
Southlake, Texas     2
San Francisco        1
New Jersey
Seattle, Washington  3
```

FIRST

Syntax

first::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions of the ORDER BY clause and OVER clause

Purpose

FIRST and LAST are very similar functions. Both are aggregate and analytic functions that operate on a set of values from a set of rows that rank as the FIRST or LAST with respect to a given sorting specification. If only one row ranks as FIRST or LAST, the aggregate operates on the set with only one element.

When you need a value from the first or last row of a sorted group, but the needed value is not the sort key, the FIRST and LAST functions eliminate the need for self joins or views and enable better performance.

- The *aggregate_function* is any one of the MIN, MAX, SUM, AVG, COUNT, VARIANCE, or STDDEV functions. It operates on values from the rows that rank either FIRST or LAST. If only one row ranks as FIRST or LAST, the aggregate operates on a singleton (nonaggregate) set.

- DENSE_RANK FIRST or DENSE_RANK LAST indicates that Oracle will aggregate over only those rows with the minimum (FIRST) or the maximum (LAST) dense rank ("olympic rank").

You can use the FIRST and LAST functions as analytic functions by specifying the OVER clause. The *query_partitioning_clause* is the only part of the OVER clause valid with these functions.

Aggregate Example

The following example returns, within each department of the sample table hr.employees, the minimum salary among the employees who make the lowest commission and the maximum salary among the employees who make the highest commission:

```
SELECT department_id,
MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct) "Worst",
MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct) "Best"
   FROM employees
   GROUP BY department_id;

DEPARTMENT_ID      Worst       Best
------------- ---------- ----------
           10       4400       4400
           20       6000      13000
           30       2500      11000
           40       6500       6500
           50       2100       8200
           60       4200       9000
           70      10000      10000
           80       6100      14000
           90      17000      24000
          100       6900      12000
          110       8300      12000
                    7000       7000
```

Analytic Example

The next example makes the same calculation as the previous example but returns the result for each employee within the department:

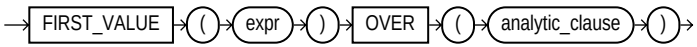
```
SELECT last_name, department_id, salary,
   MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct)
      OVER (PARTITION BY department_id) "Worst",
   MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct)
      OVER (PARTITION BY department_id) "Best"
```

```
    FROM employees
    ORDER BY department_id, salary;

LAST_NAME           DEPARTMENT_ID     SALARY      Worst       Best
------------------- ------------- ---------- ---------- ----------
Whalen                         10       4400       4400       4400
Fay                            20       6000       6000      13000
Hartstein                      20      13000       6000      13000
.
.
.
Gietz                         110       8300       8300      12000
Higgins                       110      12000       8300      12000
Grant                                   7000       7000       7000
```

FIRST_VALUE

Syntax

first_value::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

FIRST_VALUE is an analytic function. It returns the first value in an ordered set of values.

You cannot use FIRST_VALUE or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Examples

The following example selects, for each employee in Department 90, the name of the employee with the lowest salary.

```
SELECT department_id, last_name, salary, FIRST_VALUE(last_name)
```

```
  OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) AS lowest_sal
  FROM (SELECT * FROM employees WHERE department_id = 90
    ORDER BY employee_id);

DEPARTMENT_ID LAST_NAME         SALARY LOWEST_SAL
------------- ------------- ---------- -------------------------
           90 Kochhar            17000 Kochhar
           90 De Haan            17000 Kochhar
           90 King               24000 Kochhar
```

The example illustrates the nondeterministic nature of the FIRST_VALUE function. Kochhar and DeHaan have the same salary, so are in adjacent rows. Kochhar appears first because the rows returned by the subquery are ordered by employee_id. However, if the rows returned by the subquery are ordered by employee_id in descending order, as in the next example, then the function returns a different value:

```
SELECT department_id, last_name, salary, FIRST_VALUE(last_name)
  OVER (ORDER BY salary ASC ROWS UNBOUNDED PRECEDING) as fv
    FROM (SELECT * FROM employees WHERE department_id = 90
      ORDER by employee_id DESC);

DEPARTMENT_ID LAST_NAME         SALARY FV
------------- ------------- ---------- -------------------------
           90 De Haan            17000 De Haan
           90 Kochhar            17000 De Haan
           90 King               24000 De Haan
```

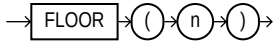
The following example shows how to make the FIRST_VALUE function deterministic by ordering on a unique key.

```
SELECT department_id, last_name, salary, hire_date,
   FIRST_VALUE(last_name) OVER
   (ORDER BY salary ASC, hire_date ROWS UNBOUNDED PRECEDING) AS fv
   FROM (SELECT * FROM employees
   WHERE department_id = 90 ORDER BY employee_id DESC);

DEPARTMENT_ID LAST_NAME         SALARY HIRE_DATE FV
------------- ------------- ---------- --------- ---------------
           90 Kochhar            17000 21-SEP-89 Kochhar
           90 De Haan            17000 13-JAN-93 Kochhar
           90 King               24000 17-JUN-87 Kochhar
```

FLOOR

Syntax

floor::=



Purpose

`FLOOR` returns largest integer equal to or less than *n*.

Examples

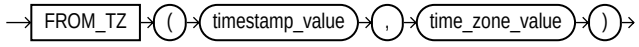
The following example returns the largest integer equal to or less than 15.7:

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

     Floor
----------
        15
```

FROM_TZ

Syntax

from_tz::=



Purpose

`FROM_TZ` converts a timestamp value at a time zone to a `TIMESTAMP WITH TIME ZONE` value. *`time_zone_value`* is a character string in the format `'TZH:TZM'` or a character expression that returns a string in `TZR` with optional `TZD` format.

Examples

The following example returns a timestamp value to `TIMESTAMP WITH TIME ZONE`:

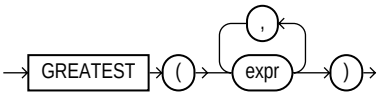
```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
   FROM DUAL;

FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
---------------------------------------------------------------
```

```
28-MAR-00 08.00.00 AM +03:00
```

GREATEST

Syntax

greatest::=



Purpose

GREATEST returns the greatest of the list of *exprs*. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher character set value. If the value returned by this function is character data, then its datatype is always VARCHAR2.

See Also: "Datatype Comparison Rules" on page 2-45

Examples

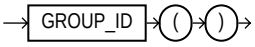
The following statement selects the string with the greatest value:

```
SELECT GREATEST ('HARRY', 'HARRIOT', 'HAROLD')
   "Greatest" FROM DUAL;

Greatest
--------
HARRY
```

GROUP_ID

Syntax

group_id::=



Purpose

GROUP_ID distinguishes duplicate groups resulting from a GROUP BY specification. It is therefore useful in filtering out duplicate groupings from the query result. It returns an Oracle NUMBER to uniquely identify duplicate groups. This function is applicable only in a SELECT statement that contains a GROUP BY clause.

If *n* duplicates exist for a particular grouping, then GROUP_ID returns numbers in the range 0 to *n*-1.

Examples

The following example assigns the value "1" to the duplicate co.country_region grouping from a query on the sample tables sh.countries and sh.sales:

```
SELECT co.country_region, co.country_subregion,
   SUM(s.amount_sold) "Revenue",
   GROUP_ID() g
FROM sales s, customers c, countries co
WHERE s.cust_id = c.cust_id AND
   c.country_id = co.country_id AND
   s.time_id = '1-JAN-00' AND
   co.country_region IN ('Americas', 'Europe')
GROUP BY co.country_region,
   ROLLUP (co.country_region, co.country_subregion);

COUNTRY_REGION       COUNTRY_SUBREGION       Revenue          G
-------------------- -------------------- ---------- ----------
Americas             Northern America         220844          0
Americas             Southern America          10872          0
Europe               Eastern Europe            12751          0
Europe               Western Europe           558686          0
Americas                                      231716          0
Europe                                        571437          0
Americas                                      231716          1
Europe                                        571437          1
```

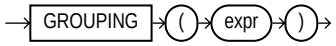
You could add the following HAVING clause to the end of the statement to ensure that only rows with GROUP_ID < 1 are returned:

```
HAVING GROUP_ID() < 1
```

GROUPING

Syntax

grouping::=



Purpose

GROUPING distinguishes superaggregate rows from regular grouped rows. GROUP BY extensions such as ROLLUP and CUBE produce superaggregate rows where the set of all values is represented by null. Using the GROUPING function, you can distinguish a null representing the set of all values in a superaggregate row from a null in a regular row.

The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 1 if the value of *expr* in the row is a null representing the set of all values. Otherwise, it returns zero. The datatype of the value returned by the GROUPING function is Oracle NUMBER.

> **See Also:** *group_by_clause* of the SELECT statement on page 18-21 for a discussion of these terms

Examples

In the following example, which uses the sample tables hr.departments and hr.employees, if the GROUPING function returns 1 (indicating a superaggregate row rather than a regular row from the table), then the string "All Jobs" appears in the "JOB" column instead of the null that would otherwise appear:

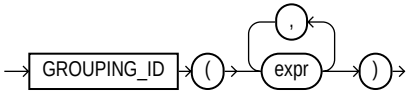
```
SELECT DECODE(GROUPING(department_name), 1, 'All Departments',
   department_name) AS department,
   DECODE(GROUPING(job_id), 1, 'All Jobs', job_id) AS job,
   COUNT(*) "Total Empl", AVG(salary) * 12 "Average Sal"
   FROM employees e, departments d
   WHERE d.department_id = e.department_id
   GROUP BY ROLLUP (department_name, job_id);

DEPARTMENT                     JOB        Total Empl Average Sal
------------------------------ ---------- ---------- -----------
Accounting                     AC_ACCOUNT          1       99600
Accounting                     AC_MGR              1      144000
Accounting                     All Jobs            2      121800
Administration                 AD_ASST             1       52800
```

```
Administration                 All Jobs            1       52800
Executive                      AD_PRES             1      288000
Executive                      AD_VP               2      204000
Executive                      All Jobs            3      232000
Finance                        FI_ACCOUNT          5       95040
Finance                        FI_MGR              1      144000
Finance                        All Jobs            6      103200
.
.
.
```

GROUPING_ID

Syntax

grouping_id::=



Purpose

GROUPING_ID returns a number corresponding to the GROUPING bit vector associated with a row. GROUPING_ID is applicable only in a SELECT statement that contains a GROUP BY extension, such as ROLLUP or CUBE, and a GROUPING function. In queries with many GROUP BY expressions, determining the GROUP BY level of a particular row requires many GROUPING functions, which leads to cumbersome SQL. GROUPING_ID is useful in these cases.

GROUPING_ID is functionally equivalent to taking the results of multiple GROUPING functions and concatenating them into a bit vector (a string of ones and zeros). By using GROUPING_ID you can avoid the need for multiple GROUPING functions and make row filtering conditions easier to express. Row filtering is easier with GROUPING_ID because the desired rows can be identified with a single condition of GROUPING_ID = *n*. The function is especially useful when storing multiple levels of aggregation in a single table.

Examples

The following example shows how to extract grouping IDs from a query of the sample table sh.sales:

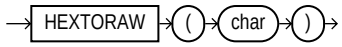
```
SELECT channel_id, promo_id, sum(amount_sold) s_sales,
   GROUPING(channel_id) gc,
   GROUPING(promo_id) gp,
```

```
   GROUPING_ID(channel_id, promo_id) gcp,
   GROUPING_ID(promo_id, channel_id) gpc
   FROM sales
   WHERE promo_id > 496
   GROUP BY CUBE(channel_id, promo_id);

C   PROMO_ID    S_SALES         GC         GP         GCP         GPC
- ---------- ---------- ---------- ---------- ---------- ----------
C        497   26094.35          0          0          0          0
C        498    22272.4          0          0          0          0
C        499    19616.8          0          0          0          0
C       9999   87781668          0          0          0          0
C            87849651.6          0          1          1          2
I        497    50325.8          0          0          0          0
I        498    52215.4          0          0          0          0
I        499   58445.85          0          0          0          0
I       9999  169497409          0          0          0          0
I            169658396          0          1          1          2
P        497   31141.75          0          0          0          0
P        498    46942.8          0          0          0          0
P        499      24156          0          0          0          0
P       9999   70890248          0          0          0          0
P            70992488.6          0          1          1          2
S        497  110629.75          0          0          0          0
S        498   82937.25          0          0          0          0
S        499   80099.15          0          0          0          0
S       9999  267205791          0          0          0          0
S            267480357          0          1          1          2
T        497     8319.6          0          0          0          0
T        498    5347.65          0          0          0          0
T        499      19781          0          0          0          0
T       9999   28095689          0          0          0          0
T            28129137.3          0          1          1          2
         497  226511.25          1          0          2          1
         498   209715.5          1          0          2          1
         499   202998.8          1          0          2          1
        9999  623470805          1          0          2          1
              624110031          1          1          3          3
```

HEXTORAW

Syntax

hextoraw::=



Purpose

HEXTORAW converts *char* containing hexadecimal digits in the CHAR, VARCHAR2, NCHAR, or NVARCHAR2 character set to a raw value.

> **Note:** This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

The following example creates a simple table with a raw column, and inserts a hexadecimal value that has been converted to RAW:

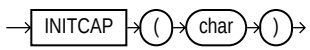
```
CREATE TABLE test (raw_col RAW(10));

INSERT INTO test VALUES (HEXTORAW('7D'));
```

> **See Also:** "RAW and LONG RAW Datatypes" on page 2-27 and RAWTOHEX on page 6-123

INITCAP

Syntax

initcap::=



Purpose

INITCAP returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

char can be of any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The return value is the same datatype as *char*.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

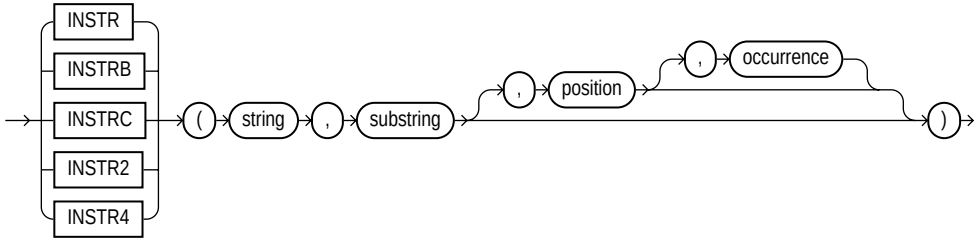
The following example capitalizes each word in the string:

```
SELECT INITCAP('the soap') "Capitals" FROM DUAL;

Capitals
---------
The Soap
```

INSTR

Syntax

instr::=



Purpose

The "in string" functions search *string* for *substring*. The function returns an integer indicating the position of the character in *string* that is the first character

of this occurrence. INSTR calculates strings using characters as defined by the input character set. INSTRB uses bytes instead of characters. INSTRC uses Unicode complete characters. INSTR2 uses UCS2 codepoints. INSTR4 uses UCS4 codepoints.

- *position* is an nonzero integer indicating the character of *string* where Oracle begins the search. If *position* is negative, then Oracle counts *and* searches backward from the end of *string*.
- *occurrence* is an integer indicating which occurrence of *string* Oracle should search for. The value of *occurrence* must be positive.

Both *string* and *substring* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The value returned is of NUMBER datatype.

The default values of both *position* and *occurrence* are 1, meaning Oracle begins searching at the first character of *string* for the first occurrence of *substring*. The return value is relative to the beginning of *string*, regardless of the value of *position*, and is expressed in characters. If the search is unsuccessful (if *substring* does not appear *occurrence* times after the *position* character of *string*), then the return value is 0.

Examples

The following example searches the string "CORPORATE FLOOR", beginning with the third character, for the string "OR". It returns the position in CORPORATE FLOOR at which the second occurrence of "OR" begins:

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2)
  "Instring" FROM DUAL;

  Instring
----------
        14
```

In the next example, Oracle counts backward from the last character to the third character from the end, which is the first "O" in "FLOOR". Oracle then searches backward for the second occurrence of OR, and finds that this second occurrence begins with the second character in the search string :

```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
     FROM DUAL;

Reversed Instring
-----------------
```

```
                2
```

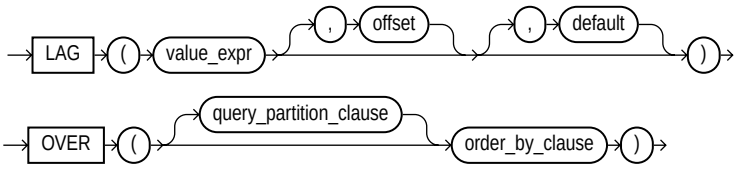
This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes"
   FROM DUAL;

Instring in bytes
-----------------
               27
```

LAG

Syntax

lag::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

LAG is an analytic function. It provides access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, LAG provides access to a row at a given physical offset prior to that position.

If you do not specify *offset*, then its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the window. If you do not specify *default*, then its default value is null.

You cannot use LAG or any other analytic function for *value_expr*. That is, you can use other built-in function expressions for *value_expr*, but you cannot nest analytic functions.

> **See Also:** "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Examples

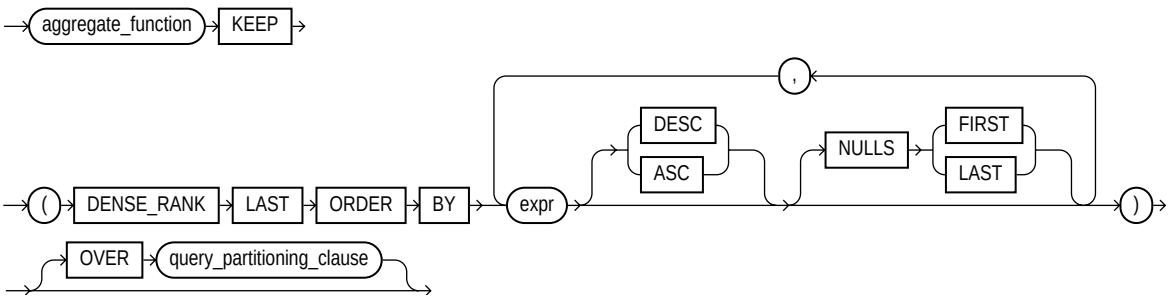
The following example provides, for each salesperson in the `employees` table, the salary of the employee hired just before:

```
SELECT last_name, hire_date, salary,
   LAG(salary, 1, 0) OVER (ORDER BY hire_date) AS prev_sal
   FROM employees
   WHERE job_id = 'PU_CLERK';

LAST_NAME                 HIRE_DATE     SALARY   PREV_SAL
------------------------- --------- ---------- ----------
Khoo                      18-MAY-95       3100          0
Tobias                    24-JUL-97       2800       3100
Baida                     24-DEC-97       2900       2800
Himuro                    15-NOV-98       2600       2900
Colmenares                10-AUG-99       2500       2600
```

LAST

Syntax

last::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions of the *query_partitioning_clause*

Purpose

`FIRST` and `LAST` are very similar functions. Both are aggregate and analytic functions that operate on a set of values from a set of rows that rank as the `FIRST` or `LAST` with respect to a given sorting specification. If only one row ranks as `FIRST` or `LAST`, the aggregate operates on the set with only one element.

When you need a value from the first or last row of a sorted group, but the needed value is not the sort key, the FIRST and LAST functions eliminate the need for self joins or views and enable better performance.

- The *aggregate_function* is any one of the MIN, MAX, SUM, AVG, COUNT, VARIANCE, or STDDEV functions. It operates on values from the rows that rank either FIRST or LAST. If only one row ranks as FIRST or LAST, the aggregate operates on a singleton (nonaggregate) set.
- DENSE_RANK FIRST or DENSE_RANK LAST indicates that Oracle will aggregate over only those rows with the minimum (FIRST) or the maximum (LAST) dense rank ("olympic rank").

You can use the FIRST and LAST functions as analytic functions by specifying the OVER clause. The *query_partitioning_clause* is the only part of the OVER clause valid with these functions.

Aggregate Example

The following example returns, within each department of the sample table hr.employees, the minimum salary among the employees who make the lowest commission and the maximum salary among the employees who make the highest commission:

```
SELECT department_id,
MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct) "Worst",
MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct) "Best"
   FROM employees
   GROUP BY department_id;

DEPARTMENT_ID      Worst       Best
------------- ---------- ----------
           10       4400       4400
           20       6000      13000
           30       2500      11000
           40       6500       6500
           50       2100       8200
           60       4200       9000
           70      10000      10000
           80       6100      14000
           90      17000      24000
          100       6900      12000
          110       8300      12000
                    7000       7000
```

Analytic Example

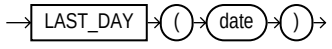
The next example makes the same calculation as the previous example but returns the result for each employee within the department:

```
SELECT last_name, department_id, salary,
   MIN(salary) KEEP (DENSE_RANK FIRST ORDER BY commission_pct)
      OVER (PARTITION BY department_id) "Worst",
   MAX(salary) KEEP (DENSE_RANK LAST ORDER BY commission_pct)
      OVER (PARTITION BY department_id) "Best"
    FROM employees
    ORDER BY department_id, salary;

LAST_NAME           DEPARTMENT_ID     SALARY      Worst        Best
------------------- ------------- ---------- ---------- ----------
Whalen                         10       4400       4400       4400
Fay                            20       6000       6000      13000
Hartstein                      20      13000       6000      13000
.
.
.
Gietz                         110       8300       8300      12000
Higgins                       110      12000       8300      12000
Grant                                   7000       7000       7000
```

LAST_DAY

Syntax

last_day::=



Purpose

LAST_DAY returns the date of the last day of the month that contains *date*.

Examples

The following statement determines how many days are left in the current month.

```
SELECT SYSDATE,
   LAST_DAY(SYSDATE) "Last",
   LAST_DAY(SYSDATE) - SYSDATE "Days Left"
   FROM DUAL;
```

```
SYSDATE   Last       Days Left
--------- --------- ----------
30-MAY-01 31-MAY-01          1
```

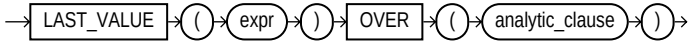
The following example adds 5 months to the hire date of each employee to give an evaluation date:

```
SELECT last_name, hire_date, TO_CHAR(
   ADD_MONTHS(LAST_DAY(hire_date), 5)) "Eval Date"
   FROM employees;

LAST_NAME                 HIRE_DATE Eval Date
------------------------- --------- ---------
King                      17-JUN-87 30-NOV-87
Kochhar                   21-SEP-89 28-FEB-90
De Haan                   13-JAN-93 30-JUN-93
Hunold                    03-JAN-90 30-JUN-90
Ernst                     21-MAY-91 31-OCT-91
Austin                    25-JUN-97 30-NOV-97
Pataballa                 05-FEB-98 31-JUL-98
Lorentz                   07-FEB-99 31-JUL-99
.
.
.
```

LAST_VALUE

Syntax

last_value::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

LAST_VALUE is an analytic function. It returns the last value in an ordered set of values.

You cannot use LAST_VALUE or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

See Also: "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Examples

The following example returns, for each row, the hire date of the employee earning the highest salary.

```
SELECT last_name, salary, hire_date, LAST_VALUE(hire_date) OVER
   (ORDER BY salary
   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM employees WHERE department_id = 90
   ORDER BY hire_date);

LAST_NAME                 SALARY HIRE_DATE LV
------------------------- ---------- --------- ---------
Kochhar                    17000 21-SEP-89 17-JUN-87
De Haan                    17000 13-JAN-93 17-JUN-87
King                       24000 17-JUN-87 17-JUN-87
```

This example illustrates the nondeterministic nature of the LAST_VALUE function. Kochhar and De Haan have the same salary, so they are in adjacent rows. Kochhar appears first because the rows in the subquery are ordered by `hire_date`. However, if the rows are ordered by `hire_date` in descending order, as in the next example, then the function returns a different value:

```
SELECT last_name, salary, hire_date, LAST_VALUE(hire_date) OVER
   (ORDER BY salary
   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM employees WHERE department_id = 90
   ORDER BY hire_date DESC);

LAST_NAME                 SALARY HIRE_DATE LV
------------------------- ---------- --------- ---------
De Haan                    17000 13-JAN-93 17-JUN-87
Kochhar                    17000 21-SEP-89 17-JUN-87
King                       24000 17-JUN-87 17-JUN-87
```

The following two examples show how to make the LAST_VALUE function deterministic by ordering on a unique key. By ordering within the function by both `salary` and `hire_date`, you can ensure the same result regardless of the ordering in the subquery.

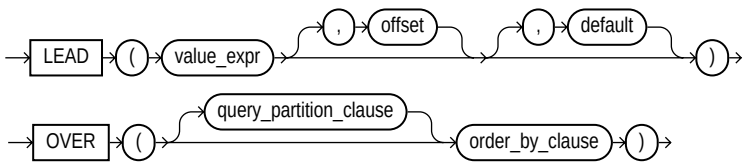
```
SELECT last_name, salary, hire_date, LAST_VALUE(hire_date) OVER
(ORDER BY salary, hire_date
```

```
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM employees WHERE department_id = 90
   ORDER BY hire_date);

LAST_NAME                    SALARY HIRE_DATE LV
------------------------- ---------- --------- ---------
Kochhar                       17000 21-SEP-89 17-JUN-87
De Haan                       17000 13-JAN-93 17-JUN-87
King                          24000 17-JUN-87 17-JUN-87

SELECT last_name, salary, hire_date, LAST_VALUE(hire_date) OVER
   (ORDER BY salary, hire_date
   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS lv
FROM (SELECT * FROM employees WHERE department_id = 90
   ORDER BY hire_date DESC);

LAST_NAME                    SALARY HIRE_DATE LV
------------------------- ---------- --------- ---------
Kochhar                       17000 21-SEP-89 17-JUN-87
De Haan                       17000 13-JAN-93 17-JUN-87
King                          24000 17-JUN-87 17-JUN-87
```

LEAD

Syntax

lead::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

LEAD is an analytic function. It provides access to more than one row of a table at the same time without a self join. Given a series of rows returned from a query and a position of the cursor, LEAD provides access to a row at a given physical offset beyond that position.

If you do not specify *offset*, then its default is 1. The optional *default* value is returned if the offset goes beyond the scope of the table. If you do not specify *default*, then its default value is null.

You cannot use LEAD or any other analytic function for *value_expr*. That is, you can use other built-in function expressions for *value_expr*, but you cannot nest analytic functions.

> **See Also:** "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Examples

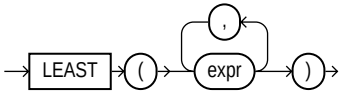
The following example provides, for each employee in the employees table, the hire date of the employee hired just after:

```
SELECT last_name, hire_date,
   LEAD(hire_date, 1) OVER (ORDER BY hire_date) AS "NextHired"
   FROM employees WHERE department_id = 30;

LAST_NAME                 HIRE_DATE NextHired
------------------------- --------- ---------
Raphaely                  07-DEC-94 18-MAY-95
Khoo                      18-MAY-95 24-JUL-97
Tobias                    24-JUL-97 24-DEC-97
Baida                     24-DEC-97 15-NOV-98
Himuro                    15-NOV-98 10-AUG-99
Colmenares                10-AUG-99
```

LEAST

Syntax

least::=



Purpose

LEAST returns the least of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares

the *expr*s using nonpadded comparison semantics. If the value returned by this function is character data, then its datatype is always VARCHAR2.

Examples

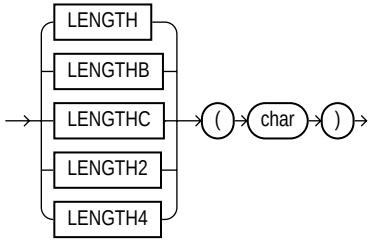
The following statement is an example of using the LEAST function:

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST"
     FROM DUAL;

LEAST
------
HAROLD
```

LENGTH

Syntax

length::=



Purpose

The "length" functions return the length of *char*. LENGTH calculates length using characters as defined by the input character set. LENGTHB uses bytes instead of characters. LENGTHC uses Unicode complete characters. LENGTH2 uses UCS2 codepoints. LENGTH4 uses UCS4 codepoints.

char can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is of datatype NUMBER. If *char* has datatype CHAR, then the length includes all trailing blanks. If *char* is null, then this function returns null.

Examples

The following example uses the LENGTH function using a single-byte database character set.

```
SELECT LENGTH('CANDIDE') "Length in characters"
   FROM DUAL;

Length in characters
--------------------
                   7
```

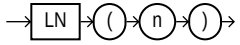
This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
   FROM DUAL;

Length in bytes
---------------
             14
```

LN

Syntax

ln::=



Purpose

LN returns the natural logarithm of *n*, where *n* is greater than 0.

Examples

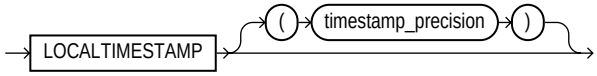
The following example returns the natural logarithm of 95:

```
SELECT LN(95) "Natural log of 95" FROM DUAL;

Natural log of 95
-----------------
       4.55387689
```

LOCALTIMESTAMP

Syntax

localtimestamp::=



Purpose

LOCALTIMESTAMP returns the current date and time in the session time zone in a value of datatype TIMESTAMP. The difference between this function and CURRENT_TIMESTAMP is that LOCALTIMESTAMP returns a TIMESTAMP value while CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.

See Also: CURRENT_TIMESTAMP on page 6-48

Examples

This example illustrates the difference between LOCALTIMESTAMP and CURRENT_TIMESTAMP:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                       LOCALTIMESTAMP
--------------------------------------------------------------------------
04-APR-00 01.27.18.999220 PM -05:00  04-APR-00 01.27.19 PM

ALTER SESSION SET TIME_ZONE = '-8:00';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;

CURRENT_TIMESTAMP                       LOCALTIMESTAMP
----------------------------------  ------------------------------
04-APR-00 10.27.45.132474 AM -08:00  04-APR-00 10.27.451 AM
```

If you use the LOCALTIMESTAMP with a format mask, take care that the format mask matches the value returned by the function. For example, consider the following table:

```
CREATE TABLE local_test (col1 TIMESTAMP WITH LOCAL TIME ZONE);
```

The following statement fails because the mask does not include the TIME ZONE portion of the return type of the function:

```
INSERT INTO local_test VALUES
   (TO_TIMESTAMP(LOCALTIMESTAMP, 'DD-MON-RR HH.MI.SSXFF'));
```

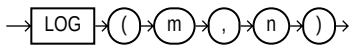
The following statement uses the correct format mask to match the return type of LOCALTIMESTAMP:

```
INSERT INTO local_test VALUES
   (TO_TIMESTAMP(LOCALTIMESTAMP, 'DD-MON-RR HH.MI.SSXFF PM'));
```

LOG

Syntax

log::=



Purpose

LOG returns the logarithm, base *m*, of *n*. The base *m* can be any positive number other than 0 or 1 and *n* can be any positive number.

Examples

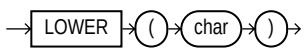
The following example returns the log of 100:

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;

Log base 10 of 100
------------------
                 2
```

LOWER

Syntax

lower::=



Purpose

LOWER returns *char*, with all letters lowercase. *char* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same datatype as *char*.

Examples

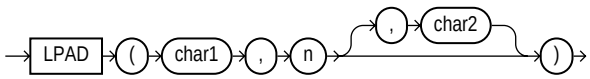
The following example returns a string in lowercase:

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
   FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax

lpad::=



Purpose

LPAD returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, then this function returns the portion of *char1* that fits in *n*.

Both *char1* and *char2* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 datatype and is in the same character set as *char1*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Examples

The following example left-pads a string with the characters "*" and ".":

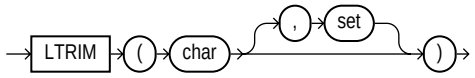
```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
```

```
   FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax

ltrim::=



Purpose

`LTRIM` removes characters from the left of *`char`*, with all the leftmost characters that appear in *`set`* removed; *`set`* defaults to a single blank. If *`char`* is a character literal, then you must enclose it in single quotes. Oracle begins scanning *`char`* from its first character and removes all characters that appear in *`set`* until reaching a character not in *`set`* and then returns the result.

Both *`char`* and *`set`* can be any of the datatypes `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` datatype and is in the same character set as *`char`*.

Examples

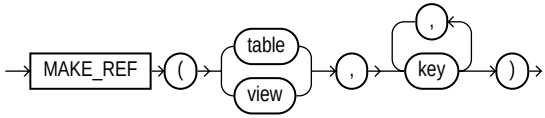
The following example trims all of the left-most x's and y's from a string:

```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
   FROM DUAL;

LTRIM example
------------
XxyLAST WORD
```

MAKE_REF

Syntax

make_ref::=



Purpose

MAKE_REF creates a REF to a row of an object view or a row in an object table whose object identifier is primary key based.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information about object views
- DEREF on page 6-56

Examples

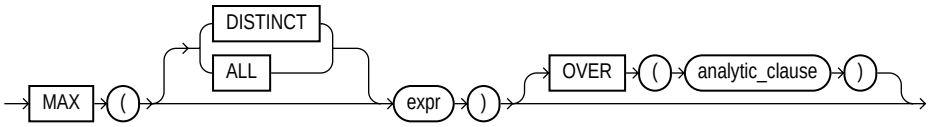
The sample schema oe contains an object view oc_inventories based on inventory_typ. The object identifier is product_id. The following example creates a REF to the row in the oc_inventories object view with a product_id of 3003:

```
SELECT MAKE_REF (oc_inventories, 3003) FROM DUAL;

MAKE_REF(OC_INVENTORIES,3003)
--------------------------------------------------------------------------
00004A038A0046857C14617141109EE03408002082543600000014260100010001
00290090606002A00078401FE0000000B03C21F040000000000000000000000000000
0000000000
```

MAX

Syntax

max::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

MAX returns maximum value of *expr*. You can use it as an aggregate or analytic function.

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example determines the highest salary in the hr.employees table:

```
SELECT MAX(salary) "Maximum" FROM employees;

   Maximum
----------
      24000
```

Analytic Example

The following example calculates, for each employee, the highest salary of the employees reporting to the same manager as the employee.

```
SELECT manager_id, last_name, salary,
   MAX(salary) OVER (PARTITION BY manager_id) AS mgr_max
```

```
   FROM employees;

MANAGER_ID LAST_NAME                 SALARY    MGR_MAX
---------- ------------------------- ---------- ----------
       100 Kochhar                    17000      17000
       100 De Haan                    17000      17000
       100 Raphaely                   11000      17000
       100 Kaufling                    7900      17000
       100 Fripp                       8200      17000
       100 Weiss                       8000      17000
.
.
.
```

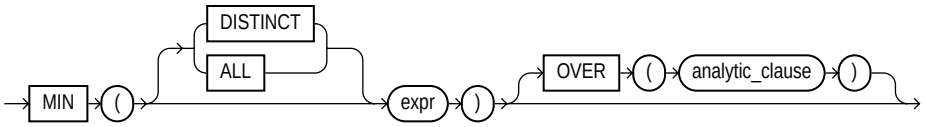
If you enclose this query in the parent query with a predicate, then you can determine the employee who makes the highest salary in each department:

```
SELECT manager_id, last_name, salary
   FROM (SELECT manager_id, last_name, salary,
      MAX(salary) OVER (PARTITION BY manager_id) AS rmax_sal
      FROM employees)
      WHERE salary = rmax_sal;

MANAGER_ID LAST_NAME                 SALARY
---------- ------------------------- ----------
       100 Kochhar                    17000
       100 De Haan                    17000
       101 Greenberg                  12000
       101 Higgens                    12000
       102 Hunold                      9000
       103 Ernst                       6000
       108 Faviet                      9000
       114 Khoo                        3100
       120 Nayer                       3200
       120 Taylor                      3200
       121 Sarchand                    4200
       122 Chung                       3800
       123 Bell                        4000
       124 Rajs                        3500
       145 Tucker                     10000
       146 King                       10000
       147 Vishney                    10500
       148 Ozer                       11500
       149 Abel                       11000
       201 Goyal                       6000
       205 Gietz                       8300
           King                       24000
```

MIN

Syntax

min::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

`MIN` returns minimum value of *`expr`*. You can use it as an aggregate or analytic function.

If you specify `DISTINCT`, then you can specify only the *`query_partition_clause`* of the *`analytic_clause`*. The *`order_by_clause`* and *`windowing_clause`* are not allowed.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *`expr`*

Aggregate Example

The following statement returns the earliest hire date in the `hr.employees` table:

```
SELECT MIN(hire_date) "Earliest" FROM employees;

Earliest
---------
17-JUN-87
```

Analytic Example

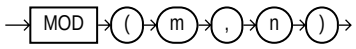
The following example determines, for each employee, the employees who were hired on or before the same date as the employee. It then determines the subset of employees reporting to the same manager as the employee, and returns the lowest salary in that subset.

```
SELECT manager_id, last_name, hire_date, salary,
   MIN(salary) OVER(PARTITION BY manager_id ORDER BY hire_date
   RANGE UNBOUNDED PRECEDING) as p_cmin
   FROM employees;

MANAGER_ID LAST_NAME                 HIRE_DATE     SALARY     P_CMIN
---------- ------------------------- --------- ---------- ----------
       100 Kochhar                   21-SEP-89      17000      17000
       100 De Haan                   13-JAN-93      17000      17000
       100 Raphaely                  07-DEC-94      11000      11000
       100 Kaufling                  01-MAY-95       7900       7900
       100 Hartstein                 17-FEB-96      13000       7900
       100 Weiss                     18-JUL-96       8000       7900
       100 Russell                   01-OCT-96      14000       7900
       100 Partners                  05-JAN-97      13500       7900
       100 Errazuriz                 10-MAR-97      12000       7900
.
.
.
```

MOD

Syntax

mod::=



Purpose

`MOD` returns the remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Examples

The following example returns the remainder of 11 divided by 4:

```
SELECT MOD(11,4) "Modulus" FROM DUAL;

   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the `MOD` function with this formula:

```
m - n * FLOOR(m/n)
```

The following table illustrates the difference between the MOD function and the classical modulus:

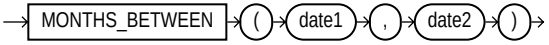
m	n	MOD(m,n)	Classical Modulus
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3

See Also: FLOOR on page 6-68

MONTHS_BETWEEN

Syntax

months_between::=



Purpose

MONTHS_BETWEEN returns number of months between dates *date1* and *date2*. If *date1* is later than *date2*, then the result is positive. If *date1* is earlier than *date2*, then the result is negative. If *date1* and *date2* are either the same days of the month or both last days of months, then the result is always an integer. Otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components *date1* and *date2*.

Examples

The following example calculates the months between two dates:

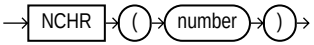
```
SELECT MONTHS_BETWEEN
   (TO_DATE('02-02-1995','MM-DD-YYYY'),
    TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
    FROM DUAL;

    Months
----------
```

```
1.03225806
```

NCHR

Syntax

nchr::=



Purpose

NCHR returns the character having the binary equivalent to *number* in the national character set. This function is equivalent to using the CHR function with the USING NCHAR_CS clause.

See Also: CHR on page 6-29

Examples

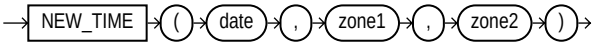
The following examples return the nchar character 187:

```
SELECT NCHR(187) FROM DUAL;

NC
--
>

SELECT CHR(187 USING NCHAR_CS) FROM DUAL;

CH
--
>
```

NEW_TIME

Syntax

new_time::=



Purpose

NEW_TIME returns the date and time in time zone *zone2* when date and time in time zone *zone1* are *date*. Before using this function, you must set the NLS_DATE_FORMAT parameter to display 24-hour time.

> **Note:** This function takes as input only a limited number of time zones. You can have access to a much greater number of time zones by combining the FROM_TZ function and the datetime expression. See FROM_TZ on page 6-68 and the example for "Datetime Expressions" on page 4-9.

The arguments *zone1* and *zone2* can be any of these text strings:

- AST, ADT: Atlantic Standard or Daylight Time
- BST, BDT: Bering Standard or Daylight Time
- CST, CDT: Central Standard or Daylight Time
- EST, EDT: Eastern Standard or Daylight Time
- GMT: Greenwich Mean Time
- HST, HDT: Alaska-Hawaii Standard Time or Daylight Time.
- MST, MDT: Mountain Standard or Daylight Time
- NST: Newfoundland Standard Time
- PST, PDT: Pacific Standard or Daylight Time
- YST, YDT: Yukon Standard or Daylight Time

Examples

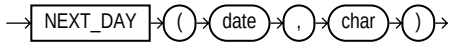
The following example returns an Atlantic Standard time, given the Pacific Standard time equivalent:

```
ALTER SESSION SET NLS_DATE_FORMAT =
   'DD-MON-YYYY HH24:MI:SS';

SELECT NEW_TIME(TO_DATE(
   '11-10-99 01:23:45', 'MM-DD-YY HH24:MI:SS'),
   'AST', 'PST') "New Date and Time" FROM DUAL;

New Date and Time
--------------------
09-NOV-1999 21:23:45
```

NEXT_DAY

Syntax

next_day::=



Purpose

NEXT_DAY returns the date of the first weekday named by *char* that is later than the date *date*. The argument *char* must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *date*.

Examples

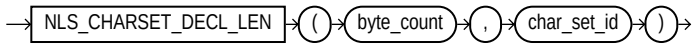
This example returns the date of the next Tuesday after February 2, 2001:

```
SELECT NEXT_DAY('02-FEB-2001','TUESDAY') "NEXT DAY"
     FROM DUAL;

NEXT DAY
-----------
06-FEB-2001
```

NLS_CHARSET_DECL_LEN

Syntax

nls_charset_decl_len::=



Purpose

NLS_CHARSET_DECL_LEN returns the declaration width (in number of characters) of an NCHAR column. The *byte_count* argument is the width of the column. The *char_set_id* argument is the character set ID of the column.

Examples

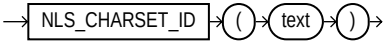
The following example returns the number of characters that are in a 200-byte column when you are using a multibyte character set:

```
SELECT NLS_CHARSET_DECL_LEN
  (200, nls_charset_id('ja16eucfixed'))
   FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
-------------------------------------------------------
                                                    100
```

NLS_CHARSET_ID

Syntax

nls_charset_id::=



Purpose

NLS_CHARSET_ID returns the character set ID number corresponding to character set name *text*. The *text* argument is a run-time VARCHAR2 value. The *text* value 'CHAR_CS' returns the database character set ID number of the server. The *text* value 'NCHAR_CS' returns the national character set ID number of the server.

Invalid character set names return null.

Examples

The following example returns the character set ID of a character set:

```
SELECT NLS_CHARSET_ID('ja16euc')
  FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
-------------------------
                      830
```

> **See Also:** *Oracle9i Database Globalization Support Guide* for a list of character set names

NLS_CHARSET_NAME

Syntax

nls_charset_name::=

Purpose

NLS_CHARSET_NAME returns the name of the character set corresponding to ID number *number*. The character set name is returned as a VARCHAR2 value in the database character set.

If *number* is not recognized as a valid character set ID, then this function returns null.

Examples

The following example returns the chartacter set corresponding to character set ID number 2:

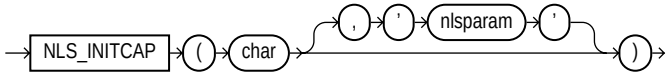
```
SELECT NLS_CHARSET_NAME(2)
  FROM DUAL;

NLS_CH
------
WE8DEC
```

See Also: *Oracle9i Database Globalization Support Guide* for a list of character set IDs

NLS_INITCAP

Syntax

nls_initcap::=



Purpose

NLS_INITCAP returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Both *char* and *'nlsparam'* can be any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The string returned is of VARCHAR2 datatype and is in the same character set as *char*.

The value of *'nlsparam'* can have this form:

```
'NLS_SORT = sort'
```

where *sort* is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. These requirements can result in a return value of a different length than the *char*. If you omit *'nlsparam'*, then this function uses the default sort sequence for your session.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

The following examples show how the linguistic sort sequence results in a different return value from the function:

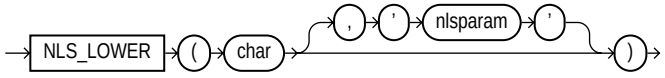
```
SELECT NLS_INITCAP
   ('ijsland') "InitCap" FROM DUAL;

InitCap
-------
Ijsland

SELECT NLS_INITCAP
   ('ijsland', 'NLS_SORT = XDutch') "InitCap"
   FROM DUAL;

InitCap
-------
IJsland
```

See Also: *Oracle9i Database Globalization Support Guide* for information on sort sequences

NLS_LOWER

Syntax

nls_lower::=



Purpose

NLS_LOWER returns *char*, with all letters lowercase.

Both *char* and *'nlsparam'* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 datatype and is in the same character set as *char*.

The *'nlsparam'* can have the same form and serve the same purpose as in the NLS_INITCAP function.

Examples

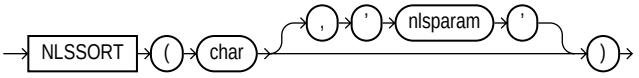
The following statement returns the character string 'citta'' using the XGerman linguistic sort sequence:

```
SELECT NLS_LOWER
   ('CITTA''', 'NLS_SORT = XGerman') "Lowercase"
   FROM DUAL;

Lowerc
------
citta'
```

NLSSORT

Syntax

nlssort::=



Purpose

NLSSORT returns the string of bytes used to sort *char*.

Both *char* and *'nlsparam'* can be any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The string returned is of RAW datatype.

The value of *'nlsparams'* can have the form

```
'NLS_SORT = sort'
```

where *sort* is a linguistic sort sequence or BINARY. If you omit *'nlsparams'*, then this function uses the default sort sequence for your session. If you specify BINARY, then this function returns *char*.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

This function can be used to specify comparisons based on a linguistic sort sequence rather than on the binary value of a string. The following example creates a test table containing two values and shows how the values returned can be ordered by the NLSSORT function:

```
CREATE TABLE test (name VARCHAR2(15));
INSERT INTO test VALUES ('Gaardiner');
INSERT INTO test VALUES ('Gaberd');

SELECT * FROM test ORDER BY name;
```

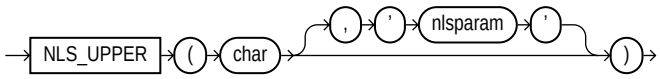
```
NAME
---------------
Gaardiner
Gaberd

SELECT * FROM test
   ORDER BY NLSSORT(name, 'NLS_SORT = XDanish');

NAME
---------------
Gaberd
Gaardiner
```

See Also: *Oracle9i Database Globalization Support Guide* for information on sort sequences

NLS_UPPER

Syntax

nls_upper::=



Purpose

NLS_UPPER returns *char*, with all letters uppercase.

Both *char* and *'nlsparam'* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 datatype and is in the same character set as *char*.

The *'nlsparam'* can have the same form and serve the same purpose as in the NLS_INITCAP function.

Examples

The following example returns a string with all the letters converted to uppercase:

```
SELECT NLS_UPPER ('große') "Uppercase"
     FROM DUAL;
```

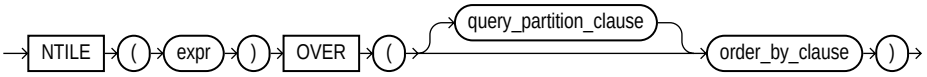
```
Upper
-----
GROßE

SELECT NLS_UPPER (’große’, ’NLS_SORT = XGerman’) "Uppercase"
   FROM DUAL;

Upperc
------
GROSSE
```

See Also: NLS_INITCAP on page 6-101

NTILE

Syntax

ntile::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

`NTILE` is an analytic function. It divides an ordered dataset into a number of buckets indicated by *`expr`* and assigns the appropriate bucket number to each row. The buckets are numbered 1 through *`expr`*, and *`expr`* must resolve to a positive constant for each partition.

The number of rows in the buckets can differ by at most 1. The remainder values (the remainder of number of rows divided by buckets) are distributed one for each bucket, starting with bucket 1.

If *`expr`* is greater than the number of rows, then a number of buckets equal to the number of rows will be filled, and the remaining buckets will be empty.

You cannot use `NTILE` or any other analytic function for *`expr`*. That is, you can use other built-in function expressions for *`expr`*, but you cannot nest analytic functions.

See Also: "About SQL Expressions" on page 4-2 for information on valid forms of *`expr`*

Examples

The following example divides into 4 buckets the values in the `salary` column of the `oe.employees` table from Department 100. The `salary` column has 6 values in this department, so the two extra values (the remainder of 6 / 4) are allocated to buckets 1 and 2, which therefore have one more value than buckets 3 or 4.

```
SELECT last_name, salary, NTILE(4) OVER (ORDER BY salary DESC)
   AS quartile FROM employees
   WHERE department_id = 100;

LAST_NAME                      SALARY   QUARTILE
------------------------- ---------- ----------
Greenberg                       12000          1
Faviet                           9000          1
Chen                             8200          2
Urman                            7800          2
Sciarra                          7700          3
Popp                             6900          4
```

NULLIF

Syntax

nullif::=

Purpose

`NULLIF` compares *`expr1`* and *`expr2`*. If they are equal, then the function returns null. If they are not equal, then the function returns *`expr1`*. You cannot specify the literal `NULL` for *`expr1`*.

The `NULLIF` function is logically equivalent to the following `CASE` expression:

```
CASE WHEN expr1 = expr 2 THEN NULL ELSE expr1 END
```

See Also: "CASE Expressions" on page 4-6

Examples

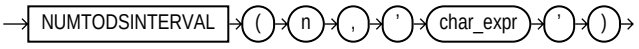
The following example selects those employees from the sample schema `hr` who have changed jobs since they were hired, as indicated by a `job_id` in the `job_history` table different from the current `job_id` in the `employees` table:

```
SELECT e.last_name, NULLIF(e.job_id, j.job_id) "Old Job ID"
   FROM employees e, job_history j
   WHERE e.employee_id = j.employee_id
   ORDER BY last_name;

LAST_NAME                 Old Job ID
------------------------- ----------
De Haan                   AD_VP
Hartstein                 MK_MAN
Kaufling                  ST_MAN
Kochhar                   AD_VP
Kochhar                   AD_VP
Raphaely                  PU_MAN
Taylor                    SA_REP
Taylor
Whalen                    AD_ASST
Whalen
```

NUMTODSINTERVAL

Syntax

numtodsinterval::=



Purpose

NUMTODSINTERVAL converts *n* to an INTERVAL DAY TO SECOND literal. *n* can be a number or an expression resolving to a number. *char_expr* can be of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype. The value for *char_expr* specifies the unit of *n* and must resolve to one of the following string values:

- 'DAY'
- 'HOUR'
- 'MINUTE'
- 'SECOND'

char_expr is case insensitive. Leading and trailing values within the parentheses are ignored. By default, precision of the return is 9.

Examples

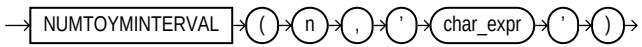
The following example calculates, for each employee, the number of employees hired by the same manager within the last 100 days from his/her hire date:

```
SELECT manager_id, last_name, hire_date,
   COUNT(*) OVER (PARTITION BY manager_id ORDER BY hire_date
   RANGE NUMTODSINTERVAL(100, 'day') PRECEDING) AS t_count
   FROM employees;

MANAGER_ID LAST_NAME                 HIRE_DATE    T_COUNT
---------- ------------------------- --------- ----------
       100 Kochhar                   21-SEP-89          1
       100 De Haan                   13-JAN-93          1
       100 Raphaely                  07-DEC-94          1
       100 Kaufling                  01-MAY-95          1
       100 Hartstein                 17-FEB-96          1
.
.
.
       149 Grant                     24-MAY-99          1
       149 Johnson                   04-JAN-00          1
       201 Goyal                     17-AUG-97          1
       205 Gietz                     07-JUN-94          1
           King                      17-JUN-87          1
```

NUMTOYMINTERVAL

Syntax

numtoyminterval::=



Purpose

NUMTOYMINTERVAL converts number *n* to an INTERVAL YEAR TO MONTH literal. *n* can be a number or an expression resolving to a number. *char_expr* can be of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype. The value for *char_expr* specifies the unit of *n*, and must resolve to one of the following string values:

- 'YEAR'
- 'MONTH'

char_expr is case insensitive. Leading and trailing values within the parentheses are ignored. By default, precision of the return is 9.

Examples

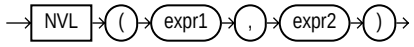
The following example calculates, for each employee, the total salary of employees hired in the past one year from his/her hire date.

```
SELECT last_name, hire_date, salary, SUM(salary)
   OVER (ORDER BY hire_date
   RANGE NUMTOYMINTERVAL(1,'year') PRECEDING) AS t_sal
   FROM employees;

LAST_NAME                 HIRE_DATE     SALARY      T_SAL
------------------------- --------- ---------- ----------
King                      17-JUN-87      24000      24000
Whalen                    17-SEP-87       4400      28400
Kochhar                   21-SEP-89      17000      17000
:
Markle                    08-MAR-00       2200     112400
Ande                      24-MAR-00       6400     106500
Banda                     21-APR-00       6200     109400
Kumar                     21-APR-00       6100     109400
```

NVL

Syntax

nvl::=



Purpose

NVL lets you replace a null (blank) with a string in the results of a query. If *expr1* is null, then NVL returns *expr2*. If *expr1* is not null, then NVL returns *expr1*. The arguments *expr1* and *expr2* can have any datatype. If their datatypes are different, then Oracle converts *expr2* to the datatype of *expr1* before comparing them.

The datatype of the return value is always the same as the datatype of *expr1*, unless *expr1* is character data, in which case the return value's datatype is VARCHAR2 and is in the character set of *expr1*.

Examples

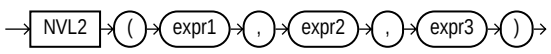
The following example returns a list of employee names and commissions, substituting "Not Applicable" if the employee receives no commission:

```
SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable')
   "COMMISSION" FROM employees
   WHERE last_name LIKE 'B%'
   ORDER BY last_name;

LAST_NAME                 COMMISSION
------------------------- ----------------------------------------
Baer                      Not Applicable
Baida                     Not Applicable
Banda                     .1
Bates                     .15
Bell                      Not Applicable
Bernstein                 .25
Bissot                    Not Applicable
Bloom                     .2
Bull                      Not Applicable
```

NVL2

Syntax

nvl2::=



Purpose

NVL2 lets you determine the value returned by a query based on whether a specified expression is null or not null. If *expr1* is not null, then NVL2 returns *expr2*. If *expr1* is null, then NVL2 returns *expr3*. The argument *expr1* can have any datatype. The arguments *expr2* and *expr3* can have any datatypes except LONG.

If the datatypes of *expr2* and *expr3* are different, then Oracle converts *expr3* to the datatype of *expr2* before comparing them unless *expr3* is a null constant. In that case, a datatype conversion is not necessary.

The datatype of the return value is always the same as the datatype of *expr2*, unless *expr2* is character data, in which case the return value's datatype is VARCHAR2.

Examples

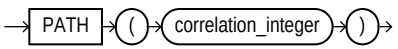
The following example shows whether the income of some employees is made up of salary plus commission, or just salary, depending on whether the `commission_pct` column of `employees` is null or not.

```
SELECT last_name, salary, NVL2(commission_pct,
   salary + (salary * commission_pct), salary) income
   FROM employees WHERE last_name like 'B%'
   ORDER BY last_name;

LAST_NAME                      SALARY     INCOME
------------------------- ---------- ----------
Baer                            10000      10000
Baida                            2900       2900
Banda                            6200       6882
Bates                            7300       8468
Bell                             4000       4000
Bernstein                        9500      11970
Bissot                           3300       3300
Bloom                           10000      12100
Bull                             4100       4100
```

PATH

Syntax

path::=



Purpose

PATH is an ancillary function used only with the UNDER_PATH and EQUALS_PATH conditions. It returns the relative path that leads to the resource specified in the parent condition.

The correlation number can be any number and is used to correlate this ancillary function with its primary condition. Values less than 1 are treated as 1.

See Also:

- EQUALS_PATH on page 5-13 UNDER_PATH on page 5-20
- the related function DEPTH on page 6-55

Examples

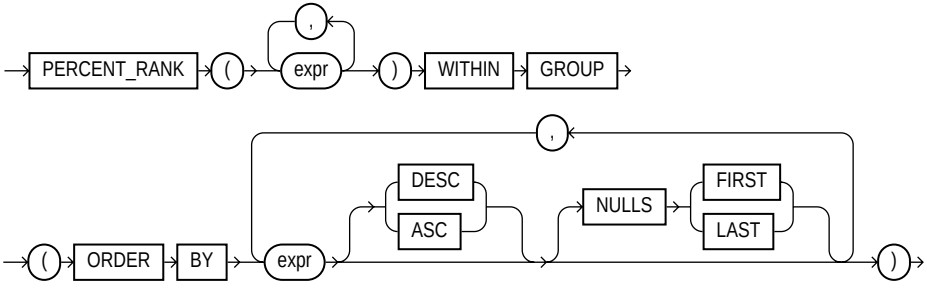
The EQUALS_PATH and UNDER_PATH conditions can take two ancillary functions, one of which is PATH. The following example shows the use of both ancillary functions. The example assumes the existence of the XMLSchema `warehouses.xsd` (created in "Using XML in SQL Statements" on page D-11).

```
SELECT PATH(1), DEPTH(2)
   FROM RESOURCE_VIEW
   WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
     AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;

PATH(1)                          DEPTH(2)
-------------------------------- --------
/www.oracle.com                         1

/www.oracle.com/xwarehouses.xsd         2
```

PERCENT_RANK

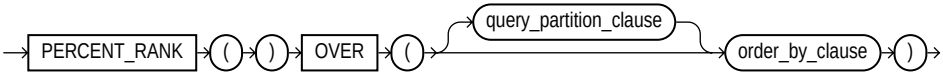
Aggregate Syntax

percent_rank_aggregate::=



Analytic Syntax

percent_rank_analytic::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

PERCENT_RANK is similar to the CUME_DIST (cumulative distribution) function. The range of values returned by PERCENT_RANK is 0 to 1, inclusive. The first row in any set has a PERCENT_RANK of 0.

- As an aggregate function, PERCENT_RANK calculates, for a hypothetical row R identified by the arguments of the function and a corresponding sort specification, the rank of row R minus 1 divided by the number of rows in the aggregate group. This calculation is made as if the hypothetical row R were inserted into the group of rows over which Oracle is to aggregate. The arguments of the function identify a single hypothetical row within each aggregate group. Therefore, they must all evaluate to constant expressions within each aggregate group. The constant argument expressions and the expressions in the ORDER BY clause of the aggregate match by position. Therefore the number of arguments must be the same and their types must be compatible.
- As an analytic function, for a row R, PERCENT_RANK calculates the rank of R minus 1, divided by 1 less than the number of rows being evaluated (the entire query result set or a partition).

Aggregate Example

The following example calculates the percent rank of a hypothetical employee in the sample table hr.employees with a salary of $15,500 and a commission of 5%:

```
SELECT PERCENT_RANK(15000, .05) WITHIN GROUP
   (ORDER BY salary, commission_pct) "Percent-Rank"
   FROM employees;

Percent-Rank
------------
  .971962617
```

Analytic Example

The following example calculates, for each employee, the percent rank of the employee's salary within the department: SELECT department_id, last_name, salary,

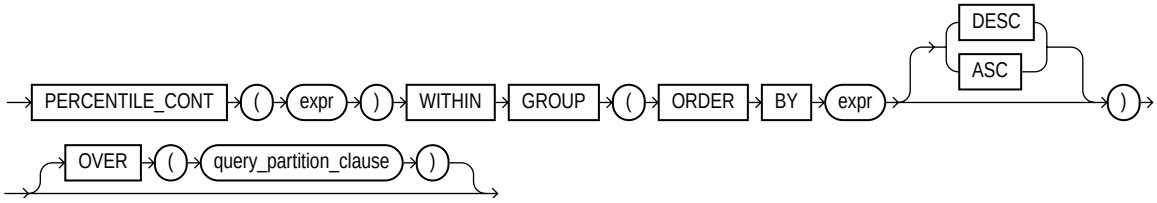
```
   PERCENT_RANK()
      OVER (PARTITION BY department_id ORDER BY salary DESC) AS pr
   FROM employees
   ORDER BY pr, salary;
```

```
DEPARTMENT_ID LAST_NAME                  SALARY         PR
------------- ------------------------- ---------- ----------
           10 Whalen                          4400          0
           40 Marvis                          6500          0
.
.
.
           80 Vishney                        10500 .176470588
           50 Everett                         3900 .181818182
           30 Khoo                            3100         .2
.
.
.
           80 Johnson                         6200 .941176471
           50 Markle                          2200 .954545455
           50 Philtanker                      2200 .954545455
           50 Olson                           2100          1
.
.
.
```

PERCENTILE_CONT

Syntax

percentile_cont::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions of the OVER clause

Purpose

PERCENTILE_CONT is an inverse distribution function that assumes a continuous distribution model. It takes a percentile value and a sort specification, and returns an interpolated value that would fall into that percentile value with respect to the sort specification. Nulls are ignored in the calculation.

The first *expr* must evaluate to a numeric value between 0 and 1, because it is a percentile value. This *expr* must be constant within each aggregation group. The ORDER BY clause takes a single expression that must be a numeric or datetime value, as these are the types over which Oracle can perform interpolation.

The result of PERCENTILE_CONT is computed by linear interpolation between values after ordering them. Using the percentile value (P) and the number of rows (N) in the aggregation group, we compute the row number we are interested in after ordering the rows with respect to the sort specification. This row number (RN) is computed according to the formula RN = (1+ (P*(N-1)). The final result of the aggregate function is computed by linear interpolation between the values from rows at row numbers CRN = CEILING(RN) and FRN = FLOOR(RN).

The final result will be:

```
if (CRN = FRN = RN) then
  (value of expression from row at RN)
else
  (CRN - RN) * (value of expression for row at FRN) +
  (RN - FRN) * (value of expression for row at CRN)
```

You can use the PERCENTILE_CONT function as an analytic function. You can specify only the *query_partitioning_clause* in its OVER clause. It returns, for each row, the value that would fall into the specified percentile among a set of values within each partition.

Aggregate Example

The following example computes the median salary in each department:

```
SELECT department_id,
   PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary DESC)
      "Median cont",
   PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY salary DESC)
      "Median disc"
   FROM employees
   GROUP BY department_id;

DEPARTMENT_ID Median-cont Median-disc
------------- ----------- -----------
           10        4400        4400
           20        9500       13000
           30        2850        2900
           40        6500        6500
           50        3100        3100
           60        4800        4800
           70       10000       10000
           80        8800        8800
           90       17000       17000
          100        8000        8200
          110       10150       12000
```

PERCENTILE_CONT and PERCENTILE_DISC may return different results. PERCENTILE_CONT returns a computed result after doing linear interpolation. PERCENTILE_DISC simply returns a value from the set of values that are aggregated over. When the percentile value is 0.5, as in this example, PERCENTILE_CONT returns the average of the two middle values for groups with even number of elements, whereas PERCENTILE_DISC returns the value of the first one among the two middle values. For aggregate groups with an odd number of elements, both functions return the value of the middle element.

Analytic Example

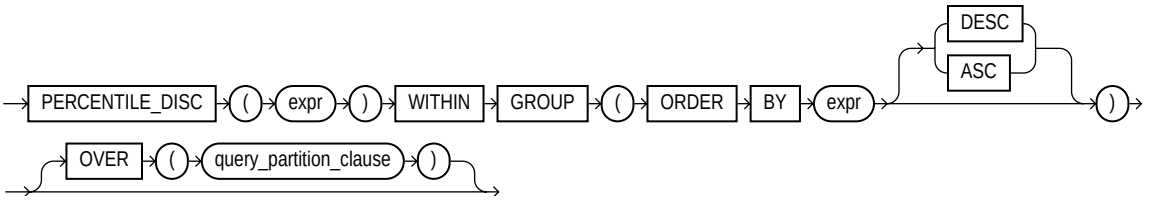
In the following example, the median for Department 60 is 4800, which has a corresponding percentile (Percent_Rank) of 0.5. None of the salaries in Department 30 have a percentile of 0.5, so the median value must be interpolated between 2900 (percentile 0.4) and 2800 (percentile 0.6), which evaluates to 2850.

```
SELECT last_name, salary, department_id,
   PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY salary DESC)
      OVER (PARTITION BY department_id) "Percentile_Cont",
   PERCENT_RANK()
      OVER (PARTITION BY department_id ORDER BY salary DESC)
"Percent_Rank"
FROM employees WHERE department_id IN (30, 60);

LAST_NAME         SALARY DEPARTMENT_ID Percentile_Cont Percent_Rank
------------- ---------- ------------- --------------- ------------
Raphaely           11000            30            2850            0
Khoo                3100            30            2850           .2
Baida               2900            30            2850           .4
Tobias              2800            30            2850           .6
Himuro              2600            30            2850           .8
Colmenares          2500            30            2850            1
Hunold              9000            60            4800            0
Ernst               6000            60            4800          .25
Austin              4800            60            4800           .5
Pataballa           4800            60            4800           .5
Lorentz             4200            60            4800            1
```

PERCENTILE_DISC

Syntax

percentile_disc::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions of the OVER clause

Purpose

PERCENTILE_DISC is an inverse distribution function that assumes a discrete distribution model. It takes a percentile value and a sort specification and returns an element from the set. Nulls are ignored in the calculation.

The first *expr* must evaluate to a numeric value between 0 and 1, because it is a percentile value. This expression must be constant within each aggregate group. The ORDER BY clause takes a single expression that can be of any type that can be sorted.

For a given percentile value P, PERCENTILE_DISC function sorts the values of the expression in the ORDER BY clause, and returns the one with the smallest CUME_DIST value (with respect to the same sort specification) that is greater than or equal to P.

Aggregate Example

See aggregate example for PERCENTILE_CONT on page 6-115.

Analytic Example

The following example calculates the median discrete percentile of the salary of each employee in the sample table hr.employees:

```
SELECT last_name, salary, department_id,
   PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY salary DESC)
      OVER (PARTITION BY department_id) "Percentile_Disc",
   CUME_DIST() OVER (PARTITION BY department_id
```

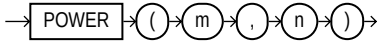
```
      ORDER BY salary DESC) "Cume_Dist"
FROM employees where department_id in (30, 60);

LAST_NAME         SALARY DEPARTMENT_ID Percentile_Disc  Cume_Dist
------------- ---------- ------------- --------------- ----------
Raphaely           11000            30            2900 .166666667
Khoo                3100            30            2900 .333333333
Baida               2900            30            2900         .5
Tobias              2800            30            2900 .666666667
Himuro              2600            30            2900 .833333333
Colmenares          2500            30            2900          1
Hunold              9000            60            4800         .2
Ernst               6000            60            4800         .4
Austin              4800            60            4800         .8
Pataballa           4800            60            4800         .8
Lorentz             4200            60            4800          1
```

The median value for Department 30 is 2900, which is the value whose corresponding percentile (`Cume_Dist`) is the smallest value greater than or equal to 0.5. The median value for Department 60 is 4800, which is the value whose corresponding percentile is the smallest value greater than or equal to 0.5.

POWER

Syntax

power::=



Purpose

`POWER` returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, then *n* must be an integer.

Examples

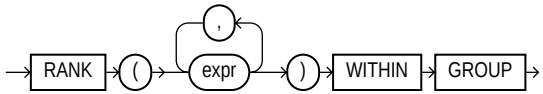
The following example returns 3 squared:

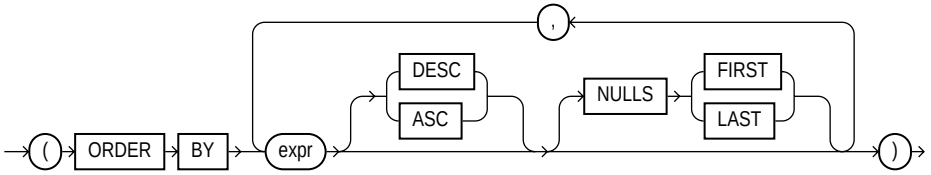
```
SELECT POWER(3,2) "Raised" FROM DUAL;

    Raised
----------
         9
```

RANK

Aggregate Syntax

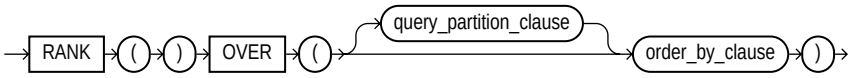
rank_aggregate::=





Analytic Syntax

rank_analytic::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

RANK calculates the rank of a value in a group of values. Rows with equal values for the ranking criteria receive the same rank. Oracle then adds the number of tied rows to the tied rank to calculate the next rank. Therefore, the ranks may not be consecutive numbers.

- As an aggregate function, RANK calculates the rank of a hypothetical row identified by the arguments of the function with respect to a given sort specification. The arguments of the function must all evaluate to constant expressions within each aggregate group, because they identify a single row within each group. The constant argument expressions and the expressions in the ORDER BY clause of the aggregate match by position. Therefore, the number of arguments must be the same and their types must be compatible.
- As an analytic function, RANK computes the rank of each row returned from a query with respect to the other rows returned by the query, based on the values of the *value_exprs* in the *order_by_clause*.

Aggregate Example

The following example calculates the rank of a hypothetical employee in the sample table `hr.employees` with a salary of $15,500 and a commission of 5%:

```
SELECT RANK(15500, .05) WITHIN GROUP
   (ORDER BY salary, commission_pct) "Rank"
   FROM employees;

      Rank
----------
       105
```

Similarly, the following query returns the rank for a $15,500 salary among the employee salaries:

```
SELECT RANK(15500) WITHIN GROUP
   (ORDER BY salary DESC) "Rank of 15500"
   FROM employees;

Rank of 15500
--------------
             4
```

Analytic Example

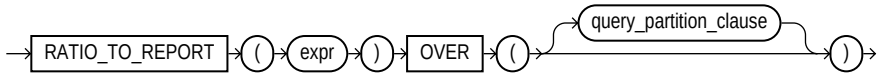
The following statement ranks the employees in the sample `hr` schema within each department based on their salary and commission. Identical salary values receive the same rank and cause nonconsecutive ranks. Compare this example with the example for DENSE_RANK on page 6-53.

```
SELECT department_id, last_name, salary, commission_pct,
   RANK() OVER (PARTITION BY department_id
   ORDER BY salary DESC, commission_pct) "Rank"
   FROM employees;

DEPARTMENT_ID LAST_NAME           SALARY COMMISSION_PCT       Rank
------------- --------------- ---------- -------------- ----------
           10 Whalen                4400                         1
           20 Hartstein            13000                         1
           20 Fay                   6000                         2
           30 Raphaely             11000                         1
           30 Khoo                  3100                         2
           30 Baida                 2900                         3
           30 Tobias                2800                         4
.
.
.
```

RATIO_TO_REPORT

Syntax

ratio_to_report::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

RATIO_TO_REPORT is an analytic function. It computes the ratio of a value to the sum of a set of values. If *expr* evaluates to null, then the ratio-to-report value also evaluates to null.

The set of values is determined by the *query_partition_clause*. If you omit that clause, then the ratio-to-report is computed over all rows returned by the query.

You cannot use RATIO_TO_REPORT or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Examples

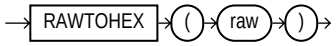
The following example calculates the ratio-to-report value of each purchasing clerk's salary to the total of all purchasing clerks' salaries:

```
SELECT last_name, salary, RATIO_TO_REPORT(salary) OVER () AS rr
   FROM employees
   WHERE job_id = 'PU_CLERK';

LAST_NAME                      SALARY         RR
------------------------- ---------- ----------
Khoo                             3100 .223021583
Baida                            2900 .208633094
Tobias                           2800 .201438849
Himuro                           2600  .18705036
Colmenares                       2500 .179856115
```

RAWTOHEX

Syntax

rawtohex::=



Purpose

RAWTOHEX converts *raw* to a character value containing its hexadecimal equivalent. The *raw* argument can be either RAW or BLOB datatype.

Examples

The following hypothetical example returns the hexadecimal equivalent of a RAW column value:

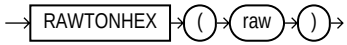
```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM graphics;

Graphics
--------
7D
```

See Also: "RAW and LONG RAW Datatypes" on page 2-27 and HEXTORAW on page 6-74

RAWTONHEX

Syntax

rawtonhex::=



Purpose

RAWTONHEX converts *raw* to an NVARCHAR2 character value containing its hexadecimal equivalent.

Examples

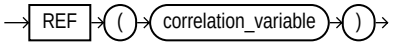
The following hypothetical example returns the hexadecimal equivalent of a RAW column value:

```
SELECT RAWTONHEX(raw_column),
   DUMP ( RAWTONHEX (raw_column) ) "DUMP"
   FROM graphics;

RAWTONHEX(RA)           DUMP
----------------------- ------------------------------
7D                      Typ=1 Len=4: 0,55,0,68
```

REF

Syntax

ref::=



Purpose

REF takes as its argument a correlation variable (table alias) associated with a row of an object table or an object view. A REF value is returned for the object instance that is bound to the variable or row.

Examples

The sample schema oe contains a type called cust_address_typ, described as follows:

```
 Attribute                           Type
 ------------------------------ ----------------
 STREET_ADDRESS                VARCHAR2(40)
 POSTAL_CODE                   VARCHAR2(10)
 CITY                          VARCHAR2(30)
 STATE_PROVINCE                VARCHAR2(10)
 COUNTRY_ID                         CHAR(2)
```

The following example creates a table based on the sample type oe.cust_address_typ, inserts a row into the table, and retrieves a REF value for the object instance of the type in the addresses table:

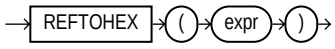
```
CREATE TABLE addresses OF cust_address_typ;

INSERT INTO addresses VALUES (
   '123 First Street', '4GF H1J', 'Our Town', 'Ourcounty', 'US');

SELECT REF(e) FROM addresses e;

REF(E)
--------------------------------------------------------------------------------
00002802097CD1261E51925B60E0340800208254367CD1261E51905B60E0340800208254360101018200
000
```

See Also: *Oracle9i Database Concepts*

REFTOHEX

Syntax

reftohex::=



Purpose

REFTOHEX converts argument *expr* to a character value containing its hexadecimal equivalent. *expr* must return a REF.

Examples

The sample schema oe contains a warehouse_typ. The following example builds on that type to illustrate how to convert the REF value of a column to a character value containing its hexadecimal equivalent:

```
CREATE TABLE warehouse_table OF warehouse_typ
   (PRIMARY KEY (warehouse_id));

CREATE TABLE location_table
   (location_number NUMBER, building REF warehouse_typ
   SCOPE IS warehouse_table);

INSERT INTO warehouse_table VALUES (1, 'Downtown', 99);

INSERT INTO location_table SELECT 10, REF(w) FROM warehouse_table w;
```

```
SELECT REFTOHEX(building) FROM location_table;

REFTOHEX(BUILDING)
--------------------------------------------------------------------------------
0000220208859B5E9255C31760E034080020825436859B5E9255C21760E034080020825436
```

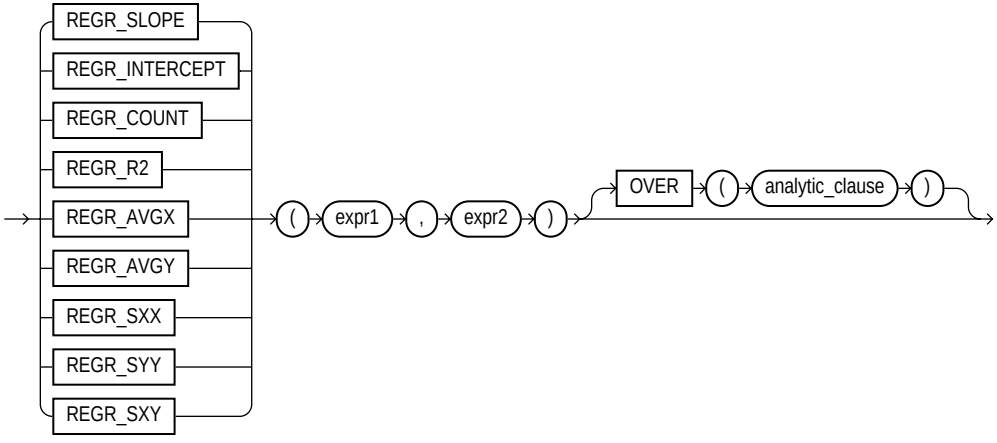
REGR_ (Linear Regression) Functions

The linear regression functions are:

- REGR_SLOPE
- REGR_INTERCEPT
- REGR_COUNT
- REGR_R2
- REGR_AVGX
- REGR_AVGY
- REGR_SXX
- REGR_SYY
- REGR_SXY

Syntax

linear_regr::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

The linear regression functions fit an ordinary-least-squares regression line to a set of number pairs. You can use them as both aggregate and analytic functions.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Oracle applies the function to the set of (*expr1*, *expr2*) pairs after eliminating all pairs for which either *expr1* or *expr2* is null. Oracle computes all the regression functions simultaneously during a single pass through the data.

expr1 is interpreted as a value of the dependent variable (a "y value"), and *expr2* is interpreted as a value of the independent variable (an "x value"). Both expressions must be numbers.

- REGR_SLOPE returns the slope of the line. The return value is a number and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  COVAR_POP(expr1, expr2) / VAR_POP(expr2)
  ```

- REGR_INTERCEPT returns the y-intercept of the regression line. The return value is a number and can be null. After the elimination of null (*expr1*, *expr2*) pairs, it makes the following computation:

  ```
  AVG(expr1) - REGR_SLOPE(expr1, expr2) * AVG(expr2)
  ```

- REGR_COUNT returns an integer that is the number of non-null number pairs used to fit the regression line.

- REGR_R2 returns the coefficient of determination (also called "R-squared" or "goodness of fit") for the regression. The return value is a number and can be null. VAR_POP(*expr1*) and VAR_POP(*expr2*) are evaluated after the elimination of null pairs. The return values are:

  ```
  NULL if VAR_POP(expr2)  = 0

     1 if VAR_POP(expr1)  = 0 and
  ```

```
                                    VAR_POP(expr2) != 0

POWER(CORR(expr1,expr),2) if VAR_POP(expr1)  > 0 and
                                    VAR_POP(expr2  != 0
```

All of the remaining regression functions return a number and can be null:

- REGR_AVGX evaluates the average of the independent variable (*expr2*) of the regression line. It makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  AVG(expr2)
  ```

- REGR_AVGY evaluates the average of the dependent variable (*expr1*) of the regression line. It makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  AVG(expr1)
  ```

REGR_SXY, REGR_SXX, REGR_SYY are auxiliary functions that are used to compute various diagnostic statistics.

- REGR_SXX makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr2)
  ```

- REGR_SYY makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * VAR_POP(expr1)
  ```

- REGR_SXY makes the following computation after the elimination of null (*expr1*, *expr2*) pairs:

  ```
  REGR_COUNT(expr1, expr2) * COVAR_POP(expr1, expr2)
  ```

The following examples are based on the sample tables `sh.sales` and `sh.products`.

General Linear Regression Example

The following example provides a comparison of the various linear regression functions:

```
SELECT
s.channel_id,
```

```
REGR_SLOPE(s.quantity_sold, p.prod_list_price)  SLOPE ,
REGR_INTERCEPT(s.quantity_sold, p.prod_list_price)  INTCPT ,
REGR_R2(s.quantity_sold, p.prod_list_price)  RSQR ,
REGR_COUNT(s.quantity_sold, p.prod_list_price)  COUNT ,
REGR_AVGX(s.quantity_sold, p.prod_list_price)  AVGLISTP ,
REGR_AVGY(s.quantity_sold, p.prod_list_price)  AVGQSOLD
FROM  sales s, products p
WHERE s.prod_id=p.prod_id AND
p.prod_category='Men'  AND
s.time_id=to_DATE('10-OCT-2000')
GROUP BY s.channel_id
;

C      SLOPE     INTCPT       RSQR      COUNT   AVGLISTP   AVGQSOLD
- ---------- ---------- ---------- ---------- ---------- ----------
C -.03529838 16.4548382 .217277422         17 87.8764706 13.3529412
I  -.0108044 13.3082392 .028398018         43  116.77907 12.0465116
P -.01729665 11.3634927 .026191191         33 80.5818182 9.96969697
S -.01277499  13.488506 .000473089         71  52.571831 12.8169014
T -.01026734 5.01019929 .064283727         21       75.2 4.23809524
```

REGR_SLOPE and REGR_INTERCEPT Examples

The following example determines the slope and intercept of the regression line for the amount of sales and sale profits for each fiscal year:

```
SELECT t.fiscal_year,
   REGR_SLOPE(s.amount_sold, s.quantity_sold) "Slope",
   REGR_INTERCEPT(s.amount_sold, s.quantity_sold) "Intercept"
   FROM sales s, times t
   WHERE s.time_id = t.time_id
   GROUP BY t.fiscal_year;

FISCAL_YEAR      Slope  Intercept
----------- ---------- ----------
       1998 49.3934247 71.6015479
       1999 49.3443482 70.1502601
       2000 49.2262135 75.0287476
```

The following example determines the cumulative slope and cumulative intercept of the regression line for the amount of and quantity of sales for two products (270 and 260) for weekend transactions (`day_number_in_week` = 6 or 7) during the last three weeks (`fiscal_week_number` of 50, 51, or 52) of 1998:

```
SELECT t.fiscal_month_number "Month", t.day_number_in_month "Day",
   REGR_SLOPE(s.amount_sold, s.quantity_sold)
```

```
   OVER (ORDER BY t.fiscal_month_desc, t.day_number_in_month) AS CUM_SLOPE,
REGR_INTERCEPT(s.amount_sold, s.quantity_sold)
   OVER (ORDER BY t.fiscal_month_desc, t.day_number_in_month) AS CUM_ICPT
FROM sales s, times t
WHERE s.time_id = t.time_id
   AND s.prod_id IN (270, 260)
   AND t.fiscal_year=1998
   AND t.fiscal_week_number IN (50, 51, 52)
   AND t.day_number_in_week IN (6,7)
ORDER BY t.fiscal_month_desc, t.day_number_in_month;

     Month        Day  CUM_SLOPE   CUM_ICPT
---------- ---------- ---------- ----------
        12         12        -68       1872
        12         12        -68       1872
        12         13 -20.244898 1254.36735
        12         13 -20.244898 1254.36735
        12         19 -18.826087       1287
        12         20 62.4561404  125.28655
        12         20 62.4561404  125.28655
        12         20 62.4561404  125.28655
        12         20 62.4561404  125.28655
        12         26 67.2658228 58.9712313
        12         26 67.2658228 58.9712313
        12         27 37.5245541 284.958221
        12         27 37.5245541 284.958221
        12         27 37.5245541 284.958221
```

REGR_COUNT Examples

The following example returns the number of customers in the `customers` table (out of a total of 319) who have account managers.

```
SELECT REGR_COUNT(customer_id, account_mgr_id) FROM customers;

REGR_COUNT(CUSTOMER_ID,ACCOUNT_MGR_ID)
--------------------------------------
                                   231
```

The following example computes the cumulative number of transactions for each day in April of 1998:

```
SELECT UNIQUE t.day_number_in_month,
   REGR_COUNT(s.amount_sold, s.quantity_sold)
      OVER (PARTITION BY t.fiscal_month_number
   ORDER BY t.day_number_in_month) "Regr_Count"
```

```
FROM sales s, times t
WHERE s.time_id = t.time_id
AND t.fiscal_year = 1998 AND t.fiscal_month_number = 4;

DAY_NUMBER_IN_MONTH Regr_Count
------------------- ----------
                  1        825
                  2       1650
                  3       2475
                  4       3300
:
:
                 26      21450
                 30      22200
```

REGR_R2 Examples

The following example computes the coefficient of determination of the regression line for amount of sales greater than 5000 and quantity sold:

```
SELECT REGR_R2(amount_sold, quantity_sold) FROM sales
   WHERE amount_sold > 5000;

REGR_R2(AMOUNT_SOLD,QUANTITY_SOLD)
----------------------------------
                          .024087453
```

The following example computes the cumulative coefficient of determination of the regression line for monthly sales amounts and quantities for each month during 1998:

```
SELECT t.fiscal_month_number,
      REGR_R2(SUM(s.amount_sold), SUM(s.quantity_sold))
      OVER (ORDER BY t.fiscal_month_number) "Regr_R2"
   FROM sales s, times t
   WHERE s.time_id = t.time_id
   AND t.fiscal_year = 1998
   GROUP BY t.fiscal_month_number
   ORDER BY t.fiscal_month_number;

FISCAL_MONTH_NUMBER    Regr_R2
------------------- ----------
                  1
                  2          1
                  3 .927372984
                  4 .807019972
```

```
 5 .932745567
 6  .94682861
 7 .965342011
 8 .955768075
 9 .959542618
10 .938618575
11 .880931415
12 .882769189
```

REGR_AVGY and REGR_AVGX Examples

The following example calculates the regression average for the amount and quantity of sales for each year:

```
SELECT t.fiscal_year,
   REGR_AVGY(s.amount_sold, s.quantity_sold) "Regr_AvgY",
   REGR_AVGX(s.amount_sold, s.quantity_sold) "Regr_AvgX"
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.fiscal_year;

FISCAL_YEAR  Regr_AvgY  Regr_AvgX
----------- ---------- ----------
       1998 716.602044 13.0584283
       1999 714.910831 13.0665536
       2000 717.331304 13.0479781
```

The following example calculates the cumulative averages for the amount and quantity of sales profits for product 260 during the last two weeks of December 1998:

```
SELECT t.day_number_in_month,
   REGR_AVGY(s.amount_sold, s.quantity_sold)
      OVER (ORDER BY t.fiscal_month_desc, t.day_number_in_month)
      "Regr_AvgY",
   REGR_AVGX(s.amount_sold, s.quantity_sold)
      OVER (ORDER BY t.fiscal_month_desc, t.day_number_in_month)
      "Regr_AvgX"
   FROM sales s, times t
   WHERE s.time_id = t.time_id
      AND s.prod_id = 260
      AND t.fiscal_month_desc = '1998-12'
      AND t.fiscal_week_number IN (51, 52)
   ORDER BY t.day_number_in_month;

DAY_NUMBER_IN_MONTH  Regr_AvgY  Regr_AvgX
```

```
------------------- ---------- ----------
                 14        882       24.5
                 14        882       24.5
                 15        801      22.25
                 15        801      22.25
                 16      777.6       21.6
                 18 642.857143 17.8571429
                 18 642.857143 17.8571429
                 20      589.5     16.375
                 21        544 15.1111111
                 22 592.363636 16.4545455
                 22 592.363636 16.4545455
                 24 553.846154 15.3846154
                 24 553.846154 15.3846154
                 26        522       14.5
                 27      578.4 16.0666667
```

REGR_SXY, REGR_SXX, and REGR_SYY Examples

The following example computes the REGR_SXY, REGR_SXX, and REGR_SYY values for the regression analysis of amount and quantity of sales for each year in the sample sh.sales table:

```
SELECT t.fiscal_year,
   REGR_SXY(s.amount_sold, s.quantity_sold) "Regr_sxy",
   REGR_SYY(s.amount_sold, s.quantity_sold) "Regr_syy",
   REGR_SXX(s.amount_sold, s.quantity_sold) "Regr_sxx"
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.fiscal_year;

FISCAL_YEAR   Regr_sxy   Regr_syy   Regr_sxx
----------- ---------- ---------- ----------
       1998 1620591607 2.3328E+11 32809865.2
       1999 1955866724 2.7695E+11 39637097.2
       2000 2127877398 3.0630E+11 43226509.7
```

The following example computes the cumulative REGR_SXY, REGR_SXX, and REGR_SYY statistics for amount and quantity of weekend sales for products 270 and 260 for each year-month value in 1998:

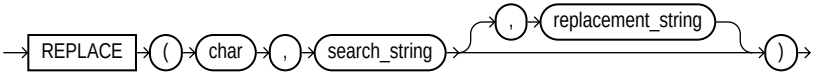
```
SELECT t.day_number_in_month,
   REGR_SXY(s.amount_sold, s.quantity_sold)
      OVER (ORDER BY t.fiscal_year, t.fiscal_month_desc) "Regr_sxy",
   REGR_SYY(s.amount_sold, s.quantity_sold)
      OVER (ORDER BY t.fiscal_year, t.fiscal_month_desc) "Regr_syy",
```

```
   REGR_SXX(s.amount_sold, s.quantity_sold)
      OVER (ORDER BY t.fiscal_year, t.fiscal_month_desc) "Regr_sxx"
FROM sales s, times t
WHERE s.time_id = t.time_id
   AND prod_id IN (270, 260)
   AND t.fiscal_month_desc = '1998-02'
   AND t.day_number_in_week IN (6,7)
ORDER BY t.day_number_in_month;

DAY_NUMBER_IN_MONTH   Regr_sxy   Regr_syy   Regr_sxx
------------------- ---------- ---------- ----------
                  1  130973783 1.8916E+10 2577797.94
                  :
                 30  130973783 1.8916E+10 2577797.94
```

REPLACE

Syntax

replace::=



Purpose

REPLACE returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, then all occurrences of *search_string* are removed. If *search_string* is null, then *char* is returned.

Both *search_string* and *replacement_string*, as well as *char*, can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 datatype and is in the same character set as *char*.

This function provides functionality related to that provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE lets you substitute one string for another as well as to remove character strings.

See Also: TRANSLATE on page 6-185

Examples

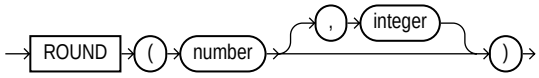
The following example replaces occurrences of "J" with "BL":

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
     FROM DUAL;

Changes
--------------
BLACK and BLUE
```

ROUND (number)

Syntax

round_number::=



Purpose

ROUND returns *number* rounded to *integer* places right of the decimal point. If *integer* is omitted, then *number* is rounded to 0 places. *integer* can be negative to round off digits left of the decimal point. *integer* must be an integer.

Examples

The following example rounds a number to one decimal point:

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2
```

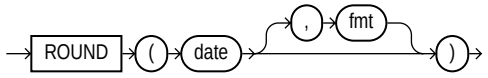
The following example rounds a number one digit to the left of the decimal point:

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;

     Round
----------
        20
```

ROUND (date)

Syntax

round_date::=



Purpose

ROUND returns *date* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, then *date* is rounded to the nearest day.

> **See Also:** "ROUND and TRUNC Date Functions" on page 6-218 for the permitted format models to use in *fmt*

Examples

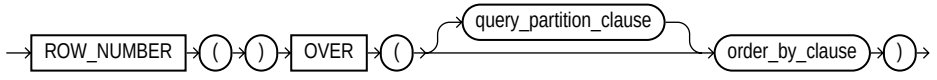
The following example rounds a date to the first day of the following year:

```
SELECT ROUND (TO_DATE ('27-OCT-00'),'YEAR')
   "New Year" FROM DUAL;

New Year
---------
01-JAN-01
```

ROW_NUMBER

Syntax

row_number::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

ROW_NUMBER is an analytic function. It assigns a unique number to each row to which it is applied (either each row in the partition or each row returned by the query), in the ordered sequence of rows specified in the *order_by_clause*, beginning with 1.

You cannot use ROW_NUMBER or any other analytic function for *expr*. That is, you can use other built-in function expressions for *expr*, but you cannot nest analytic functions.

> **See Also:** "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Examples

For each department in the sample table oe.employees, the following example assigns numbers to each row in order of employee's hire date:

```
SELECT department_id, last_name, employee_id, ROW_NUMBER()
   OVER (PARTITION BY department_id ORDER BY employee_id) AS emp_id
   FROM employees;

DEPARTMENT_ID LAST_NAME                 EMPLOYEE_ID     EMP_ID
------------- ------------------------- ----------- ----------
           10 Whalen                            200          1
           20 Hartstein                         201          1
           20 Fay                               202          2
           30 Raphaely                          114          1
           30 Khoo                              115          2
           30 Baida                             116          3
           30 Tobias                            117          4
           30 Himuro                            118          5
           30 Colmenares                        119          6
           40 Mavris                            203          1
.
.
.
          100 Popp                              113          6
          110 Higgins                           205          1
          110 Gietz                             206          2
```

ROW_NUMBER is a nondeterministic function. However, employee_id is a unique key, so the results of this application of the function are deterministic.

> **See Also:** FIRST_VALUE on page 6-66 and LAST_VALUE on page 6-81 for examples of nondeterministic behavior

ROWIDTOCHAR

Syntax

rowidtochar::=

Purpose

ROWIDTOCHAR converts a rowid value to VARCHAR2 datatype. The result of this conversion is always 18 characters long.

Examples

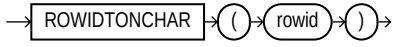
The following example converts a rowid value in the employees table to a character value. (Results vary for each build of the sample database.)

```
SELECT ROWID FROM employees
   WHERE ROWIDTOCHAR(ROWID) LIKE '%SAAb%';

ROWID
------------------
AAAFfIAAFAAAABSAAb
```

ROWIDTONCHAR

Syntax

rowidtonchar::=



Purpose

ROWIDTONCHAR converts a rowid value to NVARCHAR2 datatype. The result of this conversion is always 18 characters long.

Examples

```
SELECT LENGTHB( ROWIDTONCHAR(ROWID) ), ROWIDTONCHAR(ROWID)
   FROM employees;
```

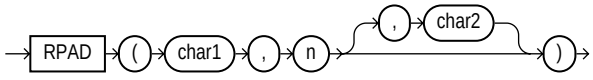
```
LENGTHB(ROWIDTONCHAR(ROWID)) ROWIDTONCHAR(ROWID
---------------------------- ------------------
                          36 AAAFfIAAFAAAABSAAA
```

⋮

RPAD

Syntax

rpad::=



Purpose

RPAD returns *char1*, right-padded to length *n* with *char2*, replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, then this function returns the portion of *char1* that fits in *n*.

Both *char1* and *char2* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 datatype and is in the same character set as *char1*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Examples

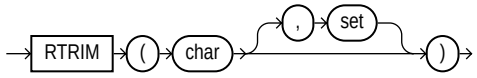
The following example right-pads a name with the letters "ab" until it is 12 characters long:

```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
     FROM DUAL;

RPAD example
------------
MORRISONabab
```

RTRIM

Syntax

rtrim::=



Purpose

RTRIM returns *char*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. If *char* is a character literal, then you must enclose it in single quotes. RTRIM works similarly to LTRIM.

Both *char* and *set* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of VARCHAR2 datatype and is in the same character set as *char*.

Examples

The following example trims the letters "xy" from the right side of a string:

```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM example"
     FROM DUAL;

RTRIM examp
-----------
BROWNINGyxX
```

See Also: LTRIM on page 6-90

SESSIONTIMEZONE

Syntax

sessiontimezone::=

Purpose

SESSIONTIMEZONE returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format '[+|]TZH:TZM') or a time

zone region name, depending on how the user specified the session time zone value in the most recent ALTER SESSION statement.

> **Note:** You can set the default client session time zone using the ORA_SDTZ environment variable. Please refer to *Oracle9i Database Globalization Support Guide* for more information on this variable.

Examples

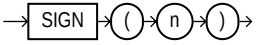
The following example returns the current session's time zone:

```
SELECT SESSIONTIMEZONE FROM DUAL;

SESSION
-------
-08:00
```

SIGN

Syntax

sign::=



Purpose

SIGN returns -1 if *n*<0, then . If *n*=0, then the function returns 0. If *n*>0, then SIGN returns 1.

Examples

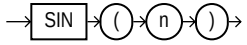
The following example indicates that the function's argument (-15) is <0:

```
SELECT SIGN(-15) "Sign" FROM DUAL;

      Sign
----------
        -1
```

SIN

Syntax

sin::=



Purpose

`SIN` returns the sine of *n* (an angle expressed in radians).

Examples

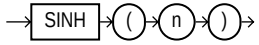
The following example returns the sin of 30 degrees:

```
SELECT SIN(30 * 3.14159265359/180)
 "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax

sinh::=



Purpose

`SINH` returns the hyperbolic sine of *n*.

Examples

The following example returns the hyperbolic sine of 1:

```
SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;

Hyperbolic sine of 1
--------------------
          1.17520119
```

SOUNDEX

Syntax

soundex::=

Purpose

SOUNDEX returns a character string containing the phonetic representation of *char*. This function lets you compare words that are spelled differently, but sound alike in English.

The phonetic representation is defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
- Assign numbers to the remaining letters (after the first) as follows:

```
b, f, p, v = 1
c, g, j, k, q, s, x, z = 2
d, t = 3
l = 4
m, n = 5
r = 6
```

- If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, then omit all but the first.
- Return the first four bytes padded with 0.

char can be of any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The return value is the same datatype as *char*.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

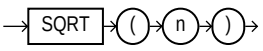
The following example returns the employees whose last names are a phonetic representation of "Smyth":

```
SELECT last_name, first_name
     FROM hr.employees
     WHERE SOUNDEX(last_name)
         = SOUNDEX('SMYTHE');

LAST_NAME  FIRST_NAME
---------- ----------
Smith      Lindsey
Smith      William
```

SQRT

Syntax

sqrt::=



Purpose

SQRT returns the square root of *n*. The value *n* cannot be negative. SQRT returns a real number.

Examples

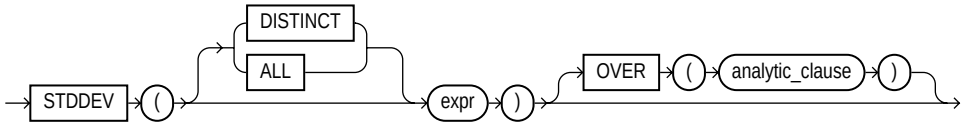
The following example returns the square root of 26:

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
5.09901951
```

STDDEV

Syntax

stddev::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

STDDEV returns sample standard deviation of *expr*, a set of numbers. You can use it as both an aggregate and analytic function. It differs from STDDEV_SAMP in that STDDEV returns zero when it has only 1 row of input data, whereas STDDEV_SAMP returns a null.

Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE aggregate function.

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 6-7, VARIANCE on page 6-203, and STDDEV_SAMP on page 6-148
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Examples

The following example returns the standard deviation of the salaries in the sample hr.employees table:

```
SELECT STDDEV(salary) "Deviation"
   FROM employees;
```

```
 Deviation
----------
3909.36575
```

Analytic Examples

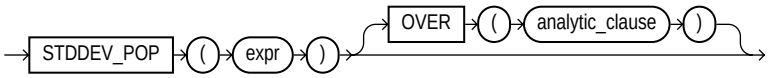
The query in the following example returns the cumulative standard deviation of the salaries in Department 80 in the sample table `hr.employees`, ordered by `hire_date`:

```
SELECT last_name, salary,
   STDDEV(salary) OVER (ORDER BY hire_date) "StdDev"
   FROM employees
   WHERE department_id = 30;

LAST_NAME                      SALARY     StdDev
------------------------- ---------- ----------
Raphaely                        11000          0
Khoo                             3100 5586.14357
Tobias                           2800  4650.0896
Baida                            2900 4035.26125
Himuro                           2600  3649.2465
Colmenares                       2500 3362.58829
```

STDDEV_POP

Syntax

stddev_pop::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

STDDEV_POP computes the population standard deviation and returns the square root of the population variance. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. This function is the same as the square root of the VAR_POP function. When VAR_POP returns null, this function returns null.

See Also:

- "Aggregate Functions" on page 6-7 and VAR_POP on page 6-199
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the population and sample standard deviations of the amount of sales in the sample table sh.sales:

```
SELECT STDDEV_POP(amount_sold) "Pop",
   STDDEV_SAMP(amount_sold) "Samp"
   FROM sales;

       Pop       Samp
---------- ----------
896.355151 896.355592
```

Analytic Example

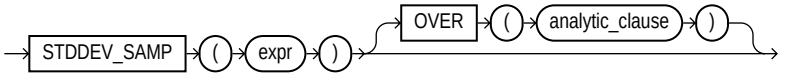
The following example returns the population standard deviations of salaries in the sample hr.employees table by department:

```
SELECT department_id, last_name, salary,
   STDDEV_POP(salary) OVER (PARTITION BY department_id) AS pop_std
   FROM employees;

DEPARTMENT_ID LAST_NAME                     SALARY    POP_STD
------------- ------------------------- ---------- ----------
           10 Whalen                          4400          0
           20 Hartstein                      13000       3500
           20 Goyal                           6000       3500
.
.
.
          100 Sciarra                         7700 1644.18166
          100 Urman                           7800 1644.18166
          100 Popp                            6900 1644.18166
          110 Higgens                        12000       1850
          110 Gietz                           8300       1850
```

STDDEV_SAMP

Syntax

stddev_samp::=



> **See Also:** "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

STDDEV_SAMP computes the cumulative sample standard deviation and returns the square root of the sample variance. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. This function is same as the square root of the VAR_SAMP function. When VAR_SAMP returns null, this function returns null.

> **See Also:**
>
> - "Aggregate Functions" on page 6-7 and VAR_SAMP on page 6-201
> - "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the population and sample standard deviations of the amount of sales in the sample table sh.sales:

```
SELECT STDDEV_POP(amount_sold) "Pop",
   STDDEV_SAMP(amount_sold) "Samp"
   FROM sales;

       Pop       Samp
---------- ----------
896.355151 896.355592
```

Analytic Example

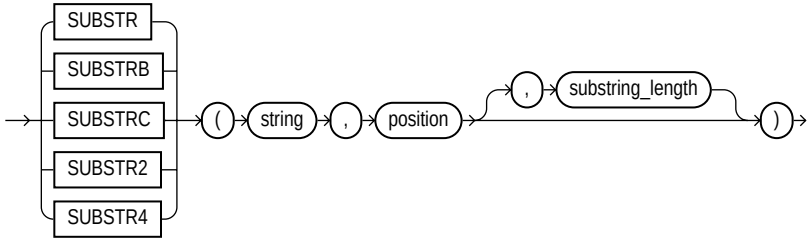
The following example returns the sample standard deviation of salaries in the `employees` table by department:

```
SELECT department_id, last_name, hire_date, salary,
   STDDEV_SAMP(salary) OVER (PARTITION BY department_id
      ORDER BY hire_date
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_sdev
   FROM employees;

DEPARTMENT_ID LAST_NAME       HIRE_DATE     SALARY   CUM_SDEV
------------- --------------- --------- ---------- ----------
           10 Whalen          17-SEP-87       4400
           20 Hartstein       17-FEB-96      13000
           20 Goyal           17-AUG-97       6000 4949.74747
           30 Raphaely        07-DEC-94      11000
           30 Khoo            18-MAY-95       3100 5586.14357
           30 Tobias          24-JUL-97       2800  4650.0896
           30 Baida           24-DEC-97       2900 4035.26125
.
.
.
          100 Chen            28-SEP-97       8200 2003.33056
          100 Sciarra         30-SEP-97       7700 1925.91969
          100 Urman           07-MAR-98       7800 1785.49713
          100 Popp            07-DEC-99       6900 1801.11077
          110 Higgens         07-JUN-94      12000
          110 Gietz           07-JUN-94       8300 2616.29509
```

SUBSTR

Syntax

substr::=



Purpose

The "substring" functions return a portion of *string*, beginning at character *position*, *substring_length* characters long. SUBSTR calculates lengths using characters as defined by the input character set. SUBSTRB uses bytes instead of characters. SUBSTRC uses Unicode complete characters. SUBSTR2 uses UCS2 codepoints. SUBSTR4 uses UCS4 codepoints.

- If *position* is 0, then it is treated as 1.
- If *position* is positive, then Oracle counts from the beginning of *string* to find the first character.
- If *position* is negative, then Oracle counts backward from the end of *string*.
- If *substring_length* is omitted, then Oracle returns all characters to the end of *string*. If *substring_length* is less than 1, then a null is returned.

string can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same datatype as *string*. Floating-point numbers passed as arguments to SUBSTR are automatically converted to integers.

Examples

The following example returns several specified substrings of "ABCDEFG":

```
SELECT SUBSTR('ABCDEFG',3,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF

SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

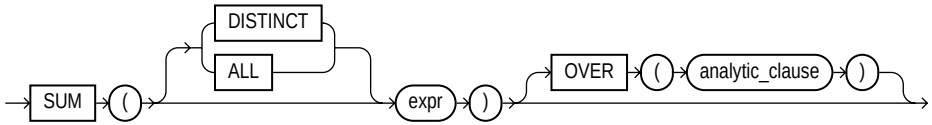
Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2) "Substring with bytes"
     FROM DUAL;

Substring with bytes
--------------------
CD
```

SUM

Syntax

sum::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

SUM returns the sum of values of *expr*. You can use it as an aggregate or analytic function.

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the sum of all salaries in the sample hr.employees table:

```
SELECT SUM(salary) "Total"
     FROM employees;

     Total
----------
    691400
```

Analytic Example

The following example calculates, for each manager in the sample table hr.employees, a cumulative total of salaries of employees who answer to that

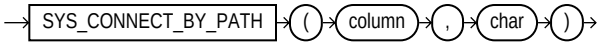
manager that are equal to or less than the current salary. You can see that Raphaely and Cambrault have the same cumulative total. This is because Raphaely and Cambrault have the identical salaries, so Oracle adds together their salary values and applies the same cumulative total to both rows.

```
SELECT manager_id, last_name, salary,
   SUM(salary) OVER (PARTITION BY manager_id ORDER BY salary
   RANGE UNBOUNDED PRECEDING) l_csum
   FROM employees;

MANAGER_ID LAST_NAME           SALARY     L_CSUM
---------- --------------- ---------- ----------
       100 Mourgos               5800       5800
       100 Vollman               6500      12300
       100 Kaufling              7900      20200
       100 Weiss                 8000      28200
       100 Fripp                 8200      36400
       100 Zlotkey              10500      46900
       100 Raphaely             11000      68900
       100 Cambrault            11000      68900
       100 Errazuriz            12000      80900
.
.
.
       149 Taylor                8600      30200
       149 Hutton                8800      39000
       149 Abel                 11000      50000
       201 Fay                   6000       6000
       205 Gietz                 8300       8300
           King                 24000      24000
```

SYS_CONNECT_BY_PATH

Syntax

sys_connect_by_path::=



Purpose

SYS_CONNECT_BY_PATH is valid only in hierarchical queries. It returns the path of a column value from root to node, with column values separated by *char* for each row returned by CONNECT BY condition.

Both *column* and *char* can be any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The string returned is of VARCHAR2 datatype and is in the same character set as *column*.

> **See Also:** "Hierarchical Queries" on page 8-3 for more information about hierarchical queries and CONNECT BY conditions

Examples

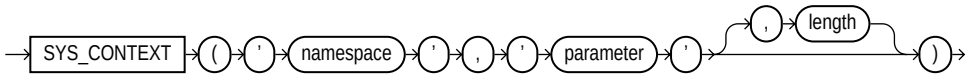
The following example returns the path of employee names from employee Kochhar to all employees of Kochhar (and their employees):

```
SELECT LPAD(' ', 2*level-1)||SYS_CONNECT_BY_PATH(last_name, '/') "Path"
   FROM employees
   START WITH last_name = 'Kochhar'
   CONNECT BY PRIOR employee_id = manager_id;

Path
--------------------------------------------------------------------
 /Kochhar
   /Kochhar/Greenberg
     /Kochhar/Greenberg/Faviet
     /Kochhar/Greenberg/Chen
     /Kochhar/Greenberg/Sciarra
     /Kochhar/Greenberg/Urman
     /Kochhar/Greenberg/Popp
   /Kochhar/Whalen
   /Kochhar/Mavris
   /Kochhar/Baer
   /Kochhar/Higgins
     /Kochhar/Higgins/Gietz
```

SYS_CONTEXT

Syntax

sys_context::=



Purpose

SYS_CONTEXT returns the value of *parameter* associated with the context *namespace*. You can use this function in both SQL and PL/SQL statements.

> **Note:** SYS_CONTEXT returns session attributes. Therefore, you cannot use it in parallel queries or in a Real Application Clusters environment.

For *namespace* and *parameter*, you can specify either a string (constant) or an expression that resolves to a string designating a namespace or an attribute. The context *namespace* must already have been created, and the associated *parameter* and its value must also have been set using the DBMS_SESSION.set_context procedure. The *namespace* must be a valid SQL identifier. The *parameter* name can be any string. It is not case sensitive, but it cannot exceed 30 bytes in length.

The datatype of the return value is VARCHAR2. The default maximum size of the return value is 256 bytes. You can override this default by specifying the optional *length* parameter. The valid range of values is 1 to 4000 bytes. (If you specify an invalid value, then Oracle ignores it and uses the default.)

Oracle9*i* provides a built-in namespace called USERENV, which describes the current session. The predefined parameters of namespace USERENV are listed Table 6–2 on page 6-155, along with the lengths of their return strings.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information on using the application context feature in your application development
- CREATE CONTEXT on page 13-12 for information on creating user-defined context namespaces
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the DBMS_SESSION.set_context procedure

Examples

The following statement returns the name of the user who logged onto the database:

```
CONNECT OE/OE
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
```

```
   FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
----------------------------------------------------
OE
```

The following hypothetical example returns the group number that was set as the value for the attribute `group_no` in the PL/SQL package that was associated with the context `hr_apps` when `hr_apps` was created:

```
SELECT SYS_CONTEXT ('hr_apps', 'group_no') "User Group"
   FROM DUAL;
```

Table 6–2 Predefined Parameters of Namespace USERENV

Parameter	Return Value	Return Length (bytes)
AUDITED_CURSORID	Returns the cursor ID of the SQL that triggered the audit.	NA
AUTHENTICATION_DATA	Data being used to authenticate the login user. For X.503 certificate authenticated sessions, this field returns the context of the certificate in HEX2 format. **Note:** You can change the return value of the AUTHENTICATION_DATA attribute using the *length* parameter of the syntax. Values of up to 4000 are accepted. This is the only attribute of USERENV for which Oracle implements such a change.	256
AUTHENTICATION_TYPE	How the user was authenticated: ■ DATABASE: user name/password authentication ■ OS: operating system external user authentication ■ NETWORK: network protocol or ANO authentication ■ PROXY: OCI proxy connection authentication	30
BG_JOB_ID	Job ID of the current session if it was established by an Oracle background process. Null if the session was not established by a background process.	64
CLIENT_IDENTIFIER	Returns the client session identifier in the global context—that is, the globally accessed application context or (in the OCI context) the OCI_ATTR_CLIENT_IDENTIFIER attribute. If no globally relevant identifier has been set, returns null.	NA

Table 6–2 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value	Return Length (bytes)
CLIENT_INFO	Returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package.	64
CURRENT_SCHEMA	Name of the default schema being used in the current schema. This value can be changed during the session with an ALTER SESSION SET CURRENT_SCHEMA statement.	30
CURRENT_SCHEMAID	Identifier of the default schema being used in the current session.	30
CURRENT_SQL	Returns the current SQL that triggered the fine-grained auditing event. You can specify this attribute only inside the event handler for the Fine-Grained Auditing feature.	64
CURRENT_USER	The name of the user whose privilege the current session is under.	30
CURRENT_USERID	User ID of the user whose privilege the current session is under.	30
DB_DOMAIN	Domain of the database as specified in the DB_DOMAIN initialization parameter.	256
DB_NAME	Name of the database as specified in the DB_NAME initialization parameter.	30
ENTRY_ID	The available auditing entry identifier. You cannot use this attribute in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.	30
EXTERNAL_NAME	External name of the database user. For SSL authenticated sessions using v.503 certificates, this field returns the distinguished name (DN) stored in the user certificate.	256
FG_JOB_ID	Job ID of the current session if it was established by a client foreground process. Null if the session was not established by a foreground process.	30
GLOBAL_CONTEXT_MEMORY	Returns the number being used in the System Global Area by the globally accessed context.	NA

Table 6–2 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value	Return Length (bytes)
HOST	Name of the host machine from which the client has connected.	54
INSTANCE	The instance identification number of the current instance.	30
IP_ADDRESS	IP address of the machine from which the client is connected.	30
ISDBA	Returns TRUE if the user has been authenticated as having DBA privileges either through the operating system or through a password file.	30
LANG	The ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.	62
LANGUAGE	The language and territory currently used by your session, along with the database character set, in this form: `language_territory.characterset`	52
NETWORK_PROTOCOL	Network protocol being used for communication, as specified in the 'PROTOCOL=*protocol*' portion of the connect string.	256
NLS_CALENDAR	The current calendar of the current session.	62
NLS_CURRENCY	The currency of the current session.	62
NLS_DATE_FORMAT	The date format for the session.	62
NLS_DATE_LANGUAGE	The language used for expressing dates.	62
NLS_SORT	BINARY or the linguistic sort basis.	62
NLS_TERRITORY	The territory of the current session.	62
OS_USER	Operating system user name of the client process that initiated the database session.	30
PROXY_USER	Name of the database user who opened the current session on behalf of SESSION_USER.	30
PROXY_USERID	Identifier of the database user who opened the current session on behalf of SESSION_USER.	30

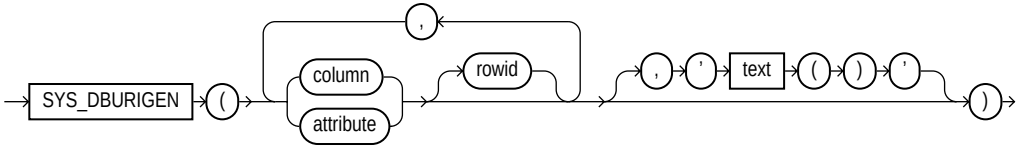
Table 6–2 (Cont.) Predefined Parameters of Namespace USERENV

Parameter	Return Value	Return Length (bytes)
SESSION_USER	Database user name by which the current user is authenticated. This value remains the same throughout the duration of the session.	30
SESSION_USERID	Identifier of the database user name by which the current user is authenticated.	30
SESSIONID	The auditing session identifier. You cannot use this attribute in distributed SQL statements.	30
TERMINAL	The operating system identifier for the client of the current session. In distributed SQL statements, this attribute returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations. (The return length of this parameter may vary by operating system.)	10

SYS_DBURIGEN

Syntax

sys_dburigen::=



Purpose

SYS_DBURIGEN takes as its argument one or more columns or attributes, and optionally a rowid, and generates a URL of datatype DBUriType to a particular column or row object. You can then use the URL to retrieve an XML document from the database.

All columns or attributes referenced must reside in the same table. They must perform the function of a primary key. That is, they need not actually match the primary keys of the table, but they must reference a unique value. If you specify

multiple columns, then all but the final column identify the row in the database, and the last column specified identifies the column within the row.

By default the URL points to a formatted XML document. If you want the URL to point only to the text of the document, then specify the optional '`text()`'. (In this XML context, the lowercase '`text`' is a keyword, not a syntactic placeholder.)

If the table or view containing the columns or attributes does not have a schema specified in the context of the query, then Oracle interprets the table or view name as a public synonym.

> **See Also:** *Oracle9i XML API Reference - XDK and Oracle XML DB* and *Oracle9i XML Developer's Kits Guide - XDK* for information on the `UriType` datatype and XML documents in the database

Examples

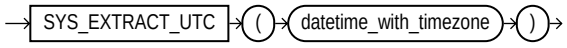
The following example uses the `SYS_DBURIGEN` function to generate a URL of datatype `DBUriType` to the `email` column of the row in the sample table `hr.employees` where the `employee_id` = 206:

```
SELECT SYS_DBURIGEN(employee_id, email)
   FROM employees
   WHERE employee_id = 206;

SYS_DBURIGEN(EMPLOYEE_ID,EMAIL)(URL, SPARE)
--------------------------------------------------------------------
DBURITYPE('/PUBLIC/EMPLOYEES/ROW[EMPLOYEE_ID=''206'']/EMAIL', NULL)
```

SYS_EXTRACT_UTC

Syntax

sys_extract_utc::=



Purpose

`SYS_EXTRACT_UTC` extracts the UTC (Coordinated Universal Time—formerly Greenwich Mean Time) from a datetime with time zone displacement.

Examples

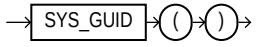
The following example extracts the UTC from a specified datetime:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2000-03-28 11:30:00.00 -08:00')
   FROM DUAL;

SYS_EXTRACT_UTC(TIMESTAMP'2000-03-2811:30:00.00-08:00')
------------------------------------------------------------------
28-MAR-00 07.30.00 PM
```

SYS_GUID

Syntax

sys_guid::=



Purpose

SYS_GUID generates and returns a globally unique identifier (RAW value) made up of 16 bytes. On most platforms, the generated identifier consists of a host identifier and a process or thread identifier of the process or thread invoking the function, and a nonrepeating value (sequence of bytes) for that process or thread.

Examples

The following example adds a column to the sample table hr.locations, inserts unique identifiers into each row, and returns the 32-character hexadecimal representation of the 16-byte RAW value of the global unique identifier:

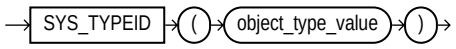
```
ALTER TABLE locations ADD (uid_col RAW(32));

UPDATE locations SET uid_col = SYS_GUID();

SELECT location_id, uid_col FROM locations;

LOCATION_ID UID_COL
----------- --------------------------------------
       1000 7CD5B7769DF75CEFE034080020825436
       1100 7CD5B7769DF85CEFE034080020825436
       1200 7CD5B7769DF95CEFE034080020825436
       1300 7CD5B7769DFA5CEFE034080020825436
:
```

.

SYS_TYPEID

Syntax

sys_typeid::=



Purpose

SYS_TYPEID returns the typeid of the most specific type of the operand. This value is used primarily to identify the type-discriminant column underlying a substitutable column. For example, you can use the value returned by SYS_TYPEID to build an index on the type-discriminant column.

Notes:

- Use this function only on object type operands.
- All final root object types—that is, final types not belonging to a type hierarchy—have a null typeid. Oracle assigns to all types belonging to a type hierarchy a unique non-null typeid.

See Also: *Oracle9i Application Developer's Guide - Object-Relational Features* for more information on typeids

Examples

The following examples use the tables persons and books, which are created in "Substitutable Table and Column Examples" on page 15-67. Both tables in turn use the person_t type, which is created in "Type Hierarchy Example" on page 16-22. The first query returns the most specific types of the object instances stored in the persons table.

```
SELECT name, SYS_TYPEID(VALUE(p)) "Type_id" FROM persons p;

NAME                      Type_id
------------------------- --------------------------------
Bob                       01
Joe                       02
Tim                       03
```

The next query returns the most specific types of authors stored in the table books:

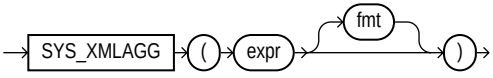
```
SELECT b.title, b.author.name, SYS_TYPEID(author)
   "Type_ID" FROM books b;

TITLE                     AUTHOR.NAME          Type_ID
------------------------- -------------------- ------------------
An Autobiography          Bob                  01
Business Rules            Joe                  02
Mixing School and Work    Tim                  03
```

You can use the SYS_TYPEID function to create an index on the type-discriminant column of a table. For an example, see "Indexing on Substitutable Columns: Examples" on page 13-89.

SYS_XMLAGG

Syntax

SYS_XMLAgg::=



Purpose

SYS_XMLAgg aggregates all of the XML documents or fragments represented by *expr* and produces a single XML document. It adds a new enclosing element with a default name ROWSET. If you want to format the XML document differently, then specify *fmt*, which is an instance of the XMLFormat object.

See Also:

- "XML Format Model" on page 2-79 for using the attributes of the XMLFormat type to format SYS_XMLAgg results
- SYS_XMLGEN on page 6-163
- *Oracle9i XML API Reference - XDK and Oracle XML DB* and *Oracle9i XML Developer's Kits Guide - XDK* for information on XML types and their use

Examples

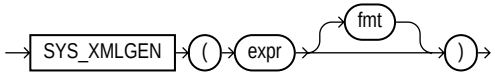
The following example uses the SYS_XMLGen function to generate an XML document for each row of the sample table employees where the employee's last name begins with the letter R, and then aggregates all of the rows into a single XML document in the default enclosing element ROWSET:

```
SELECT SYS_XMLAGG(SYS_XMLGEN(last_name))
   FROM employees
   WHERE last_name LIKE 'R%';

SYS_XMLAGG(SYS_XMLGEN(LAST_NAME))
--------------------------------------------------------------------
<ROWSET>
  <LAST_NAME>Raphaely</LAST_NAME>
  <LAST_NAME>Rogers</LAST_NAME>
  <LAST_NAME>Rajs</LAST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
</ROWSET>
```

SYS_XMLGEN

Syntax

SYS_XMLGen::=



Purpose

SYS_XMLGen takes an expression that evaluates to a particular row and column of the database, and returns an instance of type XMLType containing an XML document. The *expr* can be a scalar value, a user-defined type, or an XMLType instance.

- If *expr* is a scalar value, then the function returns an XML element containing the scalar value.
- If *expr* is a type, then the function maps the user-defined type attributes to XML elements.
- If *expr* is an XMLType instance, then the function encloses the document in an XML element whose default tag name is ROW.

By default the elements of the XML document match the elements of *expr*. For example, if *expr* resolves to a column name, then the enclosing XML element will be the same column name. If you want to format the XML document differently, then specify *fmt*, which is an instance of the XMLFormat object.

See Also:

- "XML Format Model" on page 2-79 for a description of the XMLFormat type and how to use its attributes to format SYS_XMLGen results
- *Oracle9i XML API Reference - XDK and Oracle XML DB* and *Oracle9i XML Developer's Kits Guide - XDK* for information on XML types and their use

Examples

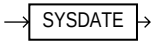
The following example retrieves the employee email ID from the sample table oe.employees where the employee_id value is 205, and generates an instance of an XMLType containing an XML document with an EMAIL element.

```
SELECT SYS_XMLGEN(email)
   FROM employees
   WHERE employee_id = 205;

SYS_XMLGEN(EMAIL)
--------------------------------------------------------------------
<EMAIL>SHIGGINS</EMAIL>
```

SYSDATE

Syntax

sysdate::=



Purpose

SYSDATE returns the current date and time. The datatype of the returned value is DATE. The function requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Examples

The following example returns the current date and time:

```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
     FROM DUAL;

NOW
-------------------
04-13-2001 09:45:51
```

SYSTIMESTAMP

Syntax

systimestamp::=

Purpose

SYSTIMESTAMP returns the system date, including fractional seconds and time zone of the system on which the database resides. The return type is TIMESTAMP WITH TIME ZONE.

Examples

The following example returns the system date.

```
SELECT SYSTIMESTAMP FROM DUAL;

SYSTIMESTAMP
---------------------------------------------------------------------------
28-MAR-00 12.38.55.538741 PM -08:00
```

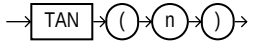
The following example shows how to explicitly specify fractional seconds:

```
SELECT TO_CHAR(SYSTIMESTAMP, 'SSSS.FF') FROM DUAL;

TO_CHAR(SYSTIM
--------------
5050.105900
```

TAN

Syntax

tan::=



Purpose

TAN returns the tangent of *n* (an angle expressed in radians).

Examples

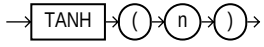
The following example returns the tangent of 135 degrees:

```
SELECT TAN(135 * 3.14159265359/180)
   "Tangent of 135 degrees"  FROM DUAL;

Tangent of 135 degrees
----------------------
                    - 1
```

TANH

Syntax

tanh::=



Purpose

TANH returns the hyperbolic tangent of *n*.

Examples

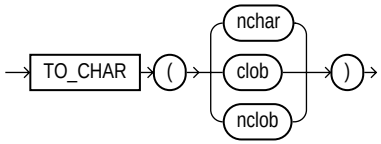
The following example returns the hyperbolic tangent of .5:

```
SELECT TANH(.5) "Hyperbolic tangent of .5"
   FROM DUAL;

Hyperbolic tangent of .5
------------------------
               .462117157
```

TO_CHAR (character)

Syntax

to_char_char::=



Purpose

TO_CHAR (character) converts NCHAR, NVARCHAR2, CLOB, or NCLOB data to the database character set.

Examples

The following example interprets a simple string as character data:

```
SELECT TO_CHAR('01110') FROM DUAL;

TO_CH
-----
01110
```

Compare this example with the first example for TO_CHAR (number) on page 6-170.

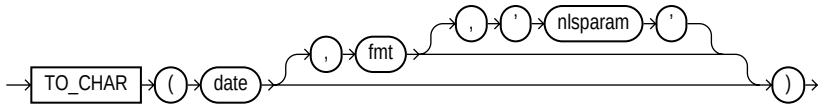
The following example converts some CLOB data from the pm.print_media table to the database character set:

```
SELECT TO_CHAR(ad_sourcetext) FROM print_media
      WHERE product_id = 2268;

TO_CHAR(AD_SOURCETEXT)
--------------------------------------------------------------------------------
*****************************
TIGER2 2268...Standard Hayes Compatible Modem
Product ID: 2268
The #1 selling modem in the universe! Tiger2's modem includes call
management and Internet voicing. Make real-time full duplex phone
calls at the same time you're online.
*********************************
```

TO_CHAR (datetime)

Syntax

to_char_date::=



Purpose

TO_CHAR (datetime) converts *date* of DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, or TIMESTAMP WITH LOCAL TIME ZONE datatype to a value of VARCHAR2 datatype in the format specified by the date format *fmt*. If you omit *fmt*, then *date* is converted to a VARCHAR2 value as follows:

- DATE is converted to a value in the default date format.
- TIMESTAMP and TIMESTAMP WITH LOCAL TIME ZONE are converted to values in the default timestamp format.
- TIMESTAMP WITH TIME ZONE is converted to a value in the default timestamp with time zone format.

The '*nlsparams*' specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit *'nlsparams'*, then this function uses the default date language for your session.

> **See Also:** "Format Models" on page 2-62 for information on date formats

Examples

The following example uses this table:

```
CREATE TABLE date_tab (
   ts_col      TIMESTAMP,
   tsltz_col   TIMESTAMP WITH LOCAL TIME ZONE,
   tstz_col    TIMESTAMP WITH TIME ZONE);
```

The example shows the results of applying TO_CHAR to different TIMESTAMP datatypes. The result for a TIMESTAMP WITH LOCAL TIME ZONE column is sensitive to session time zone, whereas the results for the TIMESTAMP and TIMESTAMP WITH TIME ZONE columns are not sensitive to session time zone:

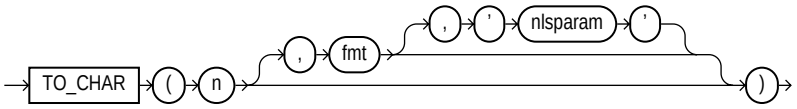
```
ALTER SESSION SET TIME_ZONE = '-8:00';
INSERT INTO date_tab VALUES (
   TIMESTAMP'1999-12-01 10:00:00',
   TIMESTAMP'1999-12-01 10:00:00',
   TIMESTAMP'1999-12-01 10:00:00');
INSERT INTO date_tab VALUES (
   TIMESTAMP'1999-12-02 10:00:00 -8:00',
   TIMESTAMP'1999-12-02 10:00:00 -8:00',
   TIMESTAMP'1999-12-02 10:00:00 -8:00');

SELECT TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF'),
   TO_CHAR(tstz_col, 'DD-MON-YYYY HH24:MI:SSxFF TZH:TZM')
   FROM date_tab;

TO_CHAR(TS_COL,'DD-MON-YYYYHH2 TO_CHAR(TSTZ_COL,'DD-MON-YYYYHH24:MI:
------------------------------ -----------------------------------
01-DEC-1999 10:00:00           01-DEC-1999 10:00:00.000000 -08:00
02-DEC-1999 10:00:00           02-DEC-1999 10:00:00.000000 -08:00

SELECT SESSIONTIMEZONE,
   TO_CHAR(tsltz_col, 'DD-MON-YYYY HH24:MI:SSxFF')
   FROM date_tab;

SESSION TO_CHAR(TSLTZ_COL,'DD-MON-YYYY
------- ------------------------------
-08:00  01-DEC-1999 10:00:00
-08:00  02-DEC-1999 10:00:00

ALTER SESSION SET TIME_ZONE = '-5:00';
SELECT TO_CHAR(ts_col, 'DD-MON-YYYY HH24:MI:SSxFF'),
   TO_CHAR(tstz_col, 'DD-MON-YYYY HH24:MI:SSxFF TZH:TZM')
   FROM date_tab;

TO_CHAR(TS_COL,'DD-MON-YYYYHH2 TO_CHAR(TSTZ_COL,'DD-MON-YYYYHH24:MI:
------------------------------ -----------------------------------
01-DEC-1999 10:00:00           01-DEC-1999 10:00:00.000000 -08:00
02-DEC-1999 10:00:00           02-DEC-1999 10:00:00.000000 -08:00
```

```
SELECT SESSIONTIMEZONE,
   TO_CHAR(tsltz_col, 'DD-MON-YYYY HH24:MI:SSxFF')
   FROM date_tab;

SESSION TO_CHAR(TSLTZ_COL,'DD-MON-YYYY
------- ----------------------------
-05:00  01-DEC-1999 13:00:00
-05:00  02-DEC-1999 13:00:00
```

TO_CHAR (number)

Syntax

to_char_number::=



Purpose

TO_CHAR (number) converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit *fmt*, then *n* is converted to a VARCHAR2 value exactly long enough to hold its significant digits.

The *'nlsparam'* specifies these characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- International currency symbol

This argument can have this form:

```
'NLS_NUMERIC_CHARACTERS = ''dg''
   NLS_CURRENCY = ''text''
   NLS_ISO_CURRENCY = territory '
```

The characters *d* and *g* represent the decimal character and group separator, respectively. They must be different single-byte characters. Note that within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit *'nlsparam'* or any one of the parameters, then this function uses the default parameter values for your session.

> **See Also:** "Format Models" on page 2-62 for information on number formats

Examples

The following statement uses implicit conversion to interpret a string and a number into a number:

```
SELECT TO_CHAR('01110' + 1) FROM dual;

TO_C
----
1111
```

Compare this example with the first example for TO_CHAR (character) on page 6-167.

In the next example, the output is blank padded to the left of the currency symbol.

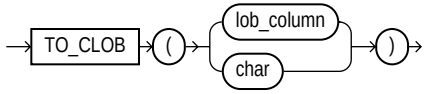
```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
     FROM DUAL;

Amount
--------------
  $10,000.00-

SELECT TO_CHAR(-10000,'L99G999D99MI',
   'NLS_NUMERIC_CHARACTERS = '',.''
   NLS_CURRENCY = ''AusDollars'' ') "Amount"
     FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

> **Note:** In the optional number format *fmt*, `L` designates local currency symbol and `MI` designates a trailing minus sign. See Table 2–13 on page 2-65 for a complete listing of number format elements.

TO_CLOB

Syntax

to_clob::=



Purpose

TO_CLOB converts NCLOB values in a LOB column or other character strings to CLOB values. *char* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. Oracle executes this function by converting the underlying LOB data from the national character set to the database character set.

Examples

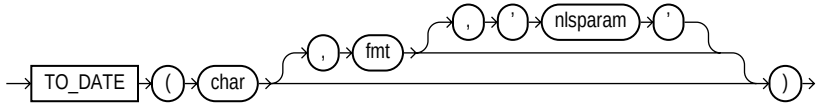
The following statement converts NCLOB data from the sample pm.print_media table to CLOB and inserts it into a CLOB column, replacing existing data in that column.

```
UPDATE PRINT_MEDIA
   SET AD_FINALTEXT = TO_CLOB (AD_FLTEXTN);
```

TO_DATE

Syntax

to_date::=



Purpose

TO_DATE converts *char* of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to a value of DATE datatype. The *fmt* is a date format specifying the format of *char*. If you omit *fmt*, then *char* must be in the default date format. If *fmt* is 'J', for Julian, then *char* must be an integer.

Note: This function does not convert data to any of the other datetime datatypes. For information on other datetime conversions, please refer to TO_TIMESTAMP on page 6-182, TO_TIMESTAMP_TZ on page 6-183, TO_DSINTERVAL on page 6-174, and "TO_YMINTERVAL" on page 6-185.

The default date format is determined implicitly by the NLS_TERRITORY initialization parameter, or can be set explicitly by the NLS_DATE_FORMAT parameter.

The *'nlsparam'* has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *char* argument. The first two digits of the returned DATE value can differ from the original *char*, depending on *fmt* or the default date format.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

See Also: "Date Format Models" on page 2-68

Examples

The following example converts a character string into a date:

```
SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;

TO_DATE('
---------
15-JAN-89
```

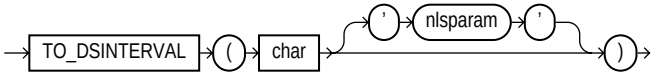
The value returned reflects the default date format if the NLS_TERRITORY parameter is set to 'AMERICA'. Different NLS_TERRITORY values result in different default date formats:

```
ALTER SESSION SET NLS_TERRITORY = 'KOREAN';

SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;

TO_DATE(
--------
89/01/15
```

TO_DSINTERVAL

Syntax

to_dsinterval::=



Purpose

TO_DSINTERVAL converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to an INTERVAL DAY TO SECOND type.

- *char* is the character string to be converted.
- The only valid *nlsparam* you can specify in this function is NLS_NUMERIC_CHARACTERS. This argument can have the form:

  ```
  NLS_NUMERIC_CHARACTERS = "dg"
  ```

  where *d* and *g* represent the decimal character and group separator respectively.

Examples

The following example selects from the employees table the employees who had worked for the company for at least 100 days on January 1, 1990:

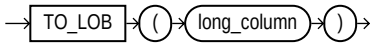
```
SELECT employee_id, last_name FROM employees
   WHERE hire_date + TO_DSINTERVAL('100 10:00:00')
   <= DATE '1990-01-01';
```

```
EMPLOYEE_ID LAST_NAME
----------- ---------------
        100 King
        101 Kochhar
        200 Whalen
```

TO_LOB

Syntax

to_LOB::=



Purpose

TO_LOB converts LONG or LONG RAW values in the column *long_column* to LOB values. You can apply this function only to a LONG or LONG RAW column, and only in the SELECT list of a subquery in an INSERT statement.

Before using this function, you must create a LOB column to receive the converted LONG values. To convert LONGs, create a CLOB column. To convert LONG RAWs, create a BLOB column.

Note: You cannot use the TO_LOB function to convert a LONG column to a LOB column in the subquery of a CREATE TABLE ...AS SELECT statement if you are creating an index-organized table. Instead, create the index-organized table without the LONG column, and then use the TO_LOB function in an INSERT ... AS SELECT statement.

See Also:

- the *modify_column_options* clause of ALTER TABLE on page 11-2 for an alternative method of converting LONG columns to LOB
- INSERT on page 17-54 for information on the subquery of an INSERT statement

Examples

The sample table `pm.print_media` has a column `press_release` of type `LONG`. This example re-creates part of the table, with LOB data in the `press_release` column:

```
CREATE TABLE new_print_media (
   product_id       NUMBER(6),
   ad_id            NUMBER(6),
   press_release    CLOB);

INSERT INTO new_print_media
   (SELECT p.product_id, p.ad_id, TO_LOB(p.press_release)
      FROM print_media p);
```

TO_MULTI_BYTE

Syntax

to_multi_byte::=

Purpose

`TO_MULTI_BYTE` returns *`char`* with all of its single-byte characters converted to their corresponding multibyte characters. *`char`* can be of datatype `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is in the same datatype as *`char`*.

Any single-byte characters in *`char`* that have no multibyte equivalents appear in the output string as single-byte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

Note: This function does not support `CLOB` data directly. However, `CLOB`s can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

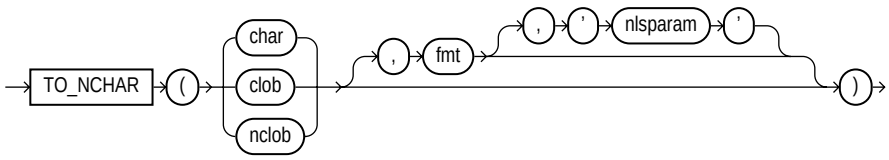
The following example illustrates converting from a single byte 'A' to a multibyte 'A' in UTF8:

```
SELECT dump(TO_MULTI_BYTE( 'A')) FROM DUAL;

DUMP(TO_MULTI_BYTE('A'))
------------------------
Typ=1 Len=3: 239,188,161
```

TO_NCHAR (character)

Syntax

to_nchar_char::=



Purpose

TO_NCHAR (character) converts a character string, CLOB, or NCLOB from the database character set to the national character set. This function is equivalent to the TRANSLATE ... USING function with a USING clause in the national character set.

> **See Also:** "Data Conversion" on page 2-48 and TRANSLATE ... USING on page 6-187

Examples

The following example converts NCLOB data from the pm.print_media table to the national character set:

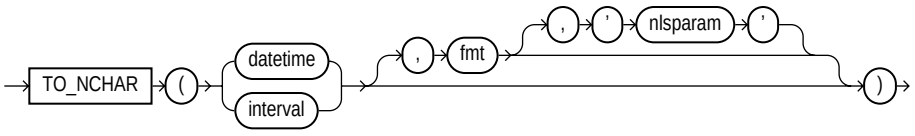
```
SELECT TO_NCHAR(ad_fltextn) FROM print_media
   WHERE product_id = 3106;

TO_NCHAR(AD_FLTEXTN)
--------------------------------------------------------------------------------
TIGER2  Tastaturen...weltweit fuehrend in Computer-Ergonomie.
TIGER2 3106 Tastatur
Product Nummer: 3106
```

```
Nur 39 EURO!
Die Tastatur KB 101/CH-DE ist eine Standard PC/AT Tastatur mit 102
Tasten. Tasta
turbelegung: Schweizerdeutsch.
. NEU: Kommt mit ergonomischer Schaumstoffunterlage.
. Extraflache und ergonimisch-geknickte Versionen verfugbar auf Anfrage.
. Lieferbar in Elfenbein, Rot oder Schwarz.
```

TO_NCHAR (datetime)

Syntax

to_nchar_date::=



Purpose

TO_NCHAR (datetime) converts a character string of DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL MONTH TO YEAR, or INTERVAL DAY TO SECOND datatype from the database character set to the national character set.

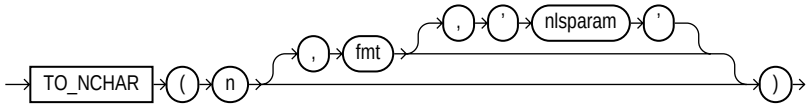
Examples

```
SELECT TO_NCHAR(order_date) FROM orders
   WHERE order_status > 9;

TO_NCHAR(ORDER_DATE)
----------------------------
14-SEP-99 08.53.40.223345 AM
13-SEP-99 09.19.00.654279 AM
27-JUN-00 08.53.32.335522 PM
26-JUN-00 09.19.43.190089 PM
06-DEC-99 01.22.34.225609 PM
```

TO_NCHAR (number)

Syntax

to_nchar_number::=



Purpose

TO_NCHAR (number) converts a number to a string in the NVARCHAR2 character set. The optional *fmt* and *'nlsparam'* corresponding to *n* can be of DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL MONTH TO YEAR, or INTERVAL DAY TO SECOND datatype.

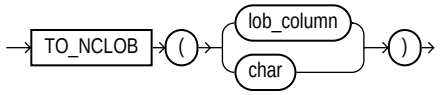
Examples

```
SELECT TO_NCHAR(customer_id) "NCHAR_Customer_ID"  FROM orders
   WHERE order_status > 9;

NCHAR_Customer_ID
-----------------
102
103
148
149
148
```

TO_NCLOB

Syntax

to_nclob::=



Purpose

TO_NCLOB converts CLOB values in a LOB column or other character strings to NCLOB values. *char* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. Oracle implements this function by converting the character set of the LOB column from the database character set to the national character set.

Examples

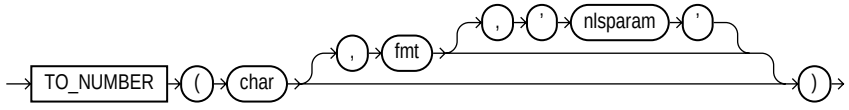
The following example inserts some character data into an NCLOB column of the pm.print_media table by first converting the data with the TO_NCLOB function:

```
INSERT INTO print_media (product_id, ad_id, ad_fltextn)
   VALUES (3502, 31001,
      TO_NCLOB('Placeholder for new product description'));
```

TO_NUMBER

Syntax

to_number::=



Purpose

TO_NUMBER converts *char*, a value of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype containing a number in the format specified by the optional format model *fmt*, to a value of NUMBER datatype.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

The following example converts character string data into a number:

```
UPDATE employees SET salary = salary +
   TO_NUMBER('100.00', '9G999D99')
   WHERE last_name = 'Perkins';
```

The *'nlsparam'* string in this function has the same purpose as it does in the TO_CHAR function for number conversions.

See Also: TO_CHAR (number) on page 6-170

```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY            = ''AusDollars''
   ') "Amount"
     FROM DUAL;

    Amount
----------
      -100
```

TO_SINGLE_BYTE

Syntax

to_single_byte::=

Purpose

TO_SINGLE_BYTE returns *char* with all of its multibyte characters converted to their corresponding single-byte characters. *char* can be of datatype CHAR,

VARCHAR2, NCHAR, or NVARCHAR2. The value returned is in the same datatype as *char*.

Any multibyte characters in *char* that have no single-byte equivalents appear in the output as multibyte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

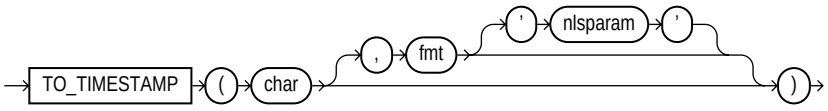
The following example illustrates going from a multibyte 'A' in UTF8 to a single byte ASCII 'A':

```
SELECT TO_SINGLE_BYTE( CHR(15711393)) FROM DUAL;

T
-
A
```

TO_TIMESTAMP

Syntax

to_timestamp::=



Purpose

TO_TIMESTAMP converts *char* of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to a value of TIMESTAMP datatype.

The optional *fmt* specifies the format of *char*. If you omit *fmt*, then *char* must be in the default format of the TIMESTAMP datatype. The optional *'nlsparam'* has the same purpose in this function as in the TO_CHAR function for date conversion.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

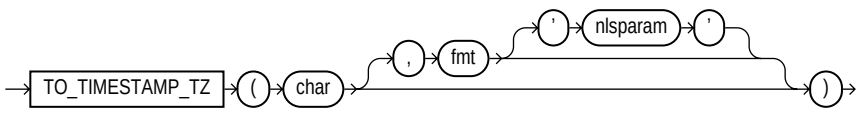
The following example converts a character string to a timestamp:

```
SELECT TO_TIMESTAMP ('1999-12-01 11:00:00', 'YYYY-MM-DD HH:MI:SS')
   FROM DUAL;

TO_TIMESTAMP('1999-12-0111:00:00','YYYY-MM-DDHH:MI:SS')
------------------------------------------------------------
01-DEC-99 11.00.00.000000000 AM
```

TO_TIMESTAMP_TZ

Syntax

to_timestamp_tz::=



Purpose

TO_TIMESTAMP_TZ converts *char* of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to a value of TIMESTAMP WITH TIME ZONE datatype.

Note: This function does not convert character strings to TIMESTAMP WITH LOCAL TIME ZONE. To do this, use a CAST function, as shown in CAST on page 6-25.

The optional *fmt* specifies the format of *char*. If you omit *fmt*, then *char* must be in the default format of the TIMESTAMP WITH TIME ZONE datatype. The optional

'nlsparam' has the same purpose in this function as in the TO_CHAR function for date conversion.

Examples

The following example converts a character string to a value of TIMESTAMP WITH TIME ZONE:

```
SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
   'YYYY-MM-DD HH:MI:SS TZH:TZM') FROM DUAL;

TO_TIMESTAMP_TZ('1999-12-0111:00:00-08:00','YYYY-MM-DDHH:MI:SSTZH:TZM')
---------------------------------------------------------------------
01-DEC-99 11.00.00.000000000 AM -08:00
```

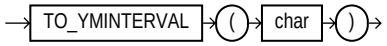
The following example casts a null column in a UNION operation as TIMESTAMP WITH LOCAL TIME ZONE using the sample tables oe.order_items and oe.orders:

```
SELECT order_id, line_item_id,
   CAST(NULL AS TIMESTAMP WITH LOCAL TIME ZONE) order_date
   FROM order_items
UNION
SELECT order_id, to_number(null), order_date
   FROM orders;

  ORDER_ID LINE_ITEM_ID ORDER_DATE
---------- ------------ -----------------------------------
      2354            1
      2354            2
      2354            3
      2354            4
      2354            5
      2354            6
      2354            7
      2354            8
      2354            9
      2354           10
      2354           11
      2354           12
      2354           13
      2354              14-JUL-00 05.18.23.234567 PM
      2355            1
      2355            2
...
```

TO_YMINTERVAL

Syntax

to_yminterval::=



Purpose

TO_YMINTERVAL converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to an INTERVAL YEAR TO MONTH type, where *char* is the character string to be converted.

Examples

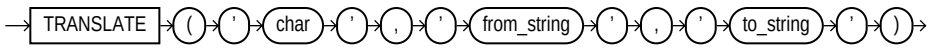
The following example calculates for each employee in the sample hr.employees table a date one year two months after the hire date:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('01-02') "14 months"
   FROM employees;

HIRE_DATE 14 months
--------- ---------
17-JUN-87 17-AUG-88
21-SEP-89 21-NOV-90
13-JAN-93 13-MAR-94
03-JAN-90 03-MAR-91
21-MAY-91 21-JUL-92
.
.
.
```

TRANSLATE

Syntax

translate::=



Purpose

TRANSLATE returns *char* with all occurrences of each character in *from_string* replaced by its corresponding character in *to_string*. Characters in *char* that are

not in *from_string* are not replaced. The argument *from_string* can contain more characters than *to_string*. In this case, the extra characters at the end of *from_string* have no corresponding characters in *to_string*. If these extra characters appear in *char*, then they are removed from the return value.

You cannot use an empty string for *to_string* to remove all characters in *from_string* from the return value. Oracle interprets the empty string as null, and if this function has a null argument, then it returns null.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012 . . . 9' are translated to '9':

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXX') "License"
     FROM DUAL;

License
--------
9XXX999
```

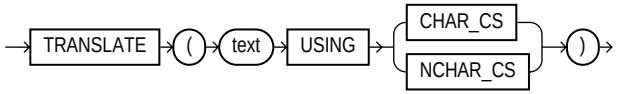
The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229',
   '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789')
   "Translate example"
     FROM DUAL;

Translate example
-----------------
2229
```

TRANSLATE ... USING

Syntax

translate_using::=



Purpose

TRANSLATE ... USING converts *text* into the character set specified for conversions between the database character set and the national character set.

> **Note:** The TRANSLATE ... USING function is supported primarily for ANSI compatibility. Oracle Corporation recommends that you use the TO_CHAR and TO_NCHAR functions, as appropriate, for converting data to the database or national character set. TO_CHAR and TO_NCHAR can take as arguments a greater variety of datatypes than TRANSLATE ... USING, which accepts only character data.

The *text* argument is the expression to be converted.

- Specifying the USING CHAR_CS argument converts *text* into the database character set. The output datatype is VARCHAR2.
- Specifying the USING NCHAR_CS argument converts *text* into the national character set. The output datatype is NVARCHAR2.

This function is similar to the Oracle CONVERT function, but must be used instead of CONVERT if either the input or the output datatype is being used as NCHAR or NVARCHAR2. If the input contains UCS2 codepoints or backslash characters (\), then use the UNISTR function.

See Also: CONVERT on page 6-34 and UNISTR on page 6-194

Examples

The following statements use data from the sample table oe.product_descriptions to show the use of the TRANSLATE ... USING function:

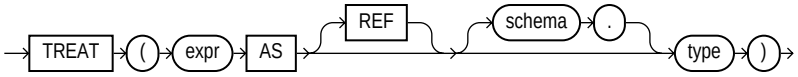
```
CREATE TABLE translate_tab (char_col  VARCHAR2(100),
                            nchar_col NVARCHAR2(50));
```

```
INSERT INTO translate_tab
   SELECT NULL, translated_name
      FROM product_descriptions
      WHERE product_id = 3501;

SELECT * FROM translate_tab;

CHAR_COL                  NCHAR_COL
------------------------- -------------------------
...
                          C per a SPNIX4.0 - Sys
                          C pro SPNIX4.0 - Sys
                          C for SPNIX4.0 - Sys
                          C til SPNIX4.0 - Sys
...

UPDATE translate_tab
   SET char_col = TRANSLATE (nchar_col USING CHAR_CS);

SELECT * FROM translate_tab;

CHAR_COL                  NCHAR_COL
------------------------- -------------------------
...
C per a SPNIX4.0 - Sys    C per a SPNIX4.0 - Sys
C pro SPNIX4.0 - Sys      C pro SPNIX4.0 - Sys
C for SPNIX4.0 - Sys      C for SPNIX4.0 - Sys
C til SPNIX4.0 - Sys      C til SPNIX4.0 - Sys
...
```

TREAT

Syntax

treat::=



Purpose

TREAT changes the declared type of an expression.

You must have the EXECUTE object privilege on *type* to use this function.

- If the declared type of *expr* is *source_type*, then type must be some supertype or subtype of *source_type*. If the most specific type of *expr* is *type* (or some subtype of *type*), then TREAT returns *expr*. If the most specific type of *expr* is not *type* (or some subtype of *type*), then TREAT returns NULL.
- If the declared type of *expr* is REF *source_type*, then *type* must be some subtype or supertype of *source_type*. If the most specific type of DEREF(*expr*) is *type* (or a subtype of *type*), then TREAT returns *expr*. If the most specific type of DEREF(*expr*) is not *type* (or a subtype of *type*), then TREAT returns NULL.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

The following statement uses the table oe.persons, which is created in "Substitutable Table and Column Examples" on page 15-67. That table is based on the person_t type, which is created in "Type Hierarchy Example" on page 16-22. The example retrieves the salary attribute of all people in the persons table, the value being null for instances of people that are not employees.

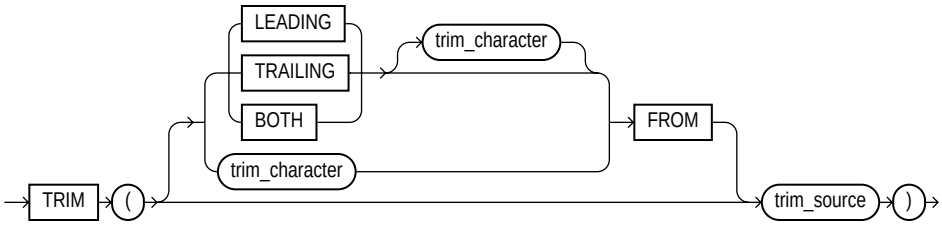
```
SELECT name, TREAT(VALUE(p) AS employee_t).salary salary
   FROM persons p;

NAME                           SALARY
------------------------ ----------
Bob
Joe                            100000
Tim                              1000
```

You can use the TREAT function to create an index on the subtype attributes of a substitutable column. For an example, see "Indexing on Substitutable Columns: Examples" on page 13-89.

TRIM

Syntax

trim::=



Purpose

`TRIM` enables you to trim leading or trailing characters (or both) from a character string. If *`trim_character`* or *`trim_source`* is a character literal, then you must enclose it in single quotes.

- If you specify `LEADING`, then Oracle removes any leading characters equal to *`trim_character`*.
- If you specify `TRAILING`, then Oracle removes any trailing characters equal to *`trim_character`*.
- If you specify `BOTH` or none of the three, then Oracle removes leading and trailing characters equal to *`trim_character`*.
- If you do not specify *`trim_character`*, then the default value is a blank space.
- If you specify only *`trim_source`*, then Oracle removes leading and trailing blank spaces.
- The function returns a value with datatype `VARCHAR2`. The maximum length of the value is the length of *`trim_source`*.
- If either *`trim_source`* or *`trim_character`* is null, then the `TRIM` function returns null.

Both *`trim_character`* and *`trim_source`* can be any of the datatypes `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of `VARCHAR2` datatype and is in the same character set as *`trim_source`*.

Examples

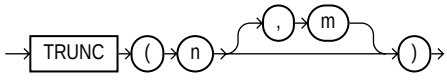
This example trims leading and trailing zeroes from a number:

```
SELECT TRIM (0 FROM 0009872348900) "TRIM Example"
   FROM DUAL;

TRIM Example
------------
    98723489
```

TRUNC (number)

Syntax

trunc_number::=



Purpose

The TRUNC (number) function returns *n* truncated to *m* decimal places. If *m* is omitted, then *n* is truncated to 0 places. *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

Examples

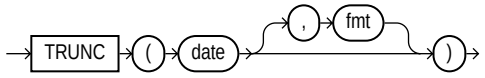
The following example truncate numbers:

```
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

  Truncate
----------
      15.7


SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

  Truncate
----------
        10
```

TRUNC (date)

Syntax

trunc_date::=



Purpose

The TRUNC (date) function returns *date* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, then *date* is truncated to the nearest day.

> **See Also:** "ROUND and TRUNC Date Functions" on page 6-218 for the permitted format models to use in *fmt*

Examples

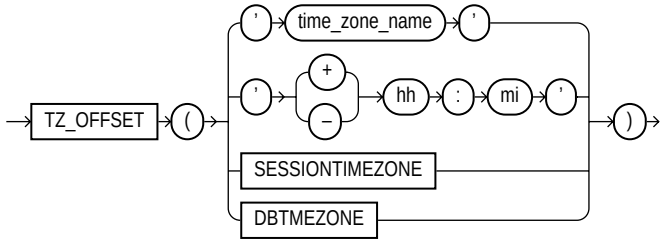
The following example truncates a date:

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
  "New Year" FROM DUAL;

New Year
---------
01-JAN-92
```

TZ_OFFSET

Syntax

tz_offset::=



Purpose

TZ_OFFSET returns the time zone offset corresponding to the value entered based on the date the statement is executed. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword SESSIONTIMEZONE or DBTIMEZONE. For a listing of valid values, query the TZNAME column of the V$TIMEZONE_NAMES dynamic performance view.

> **See Also:** *Oracle9i Database Reference* for information on the dynamic performance views

Examples

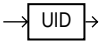
The following example returns the time zone offset of the US/Eastern time zone from UTC:

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;

TZ_OFFS
-------
-04:00
```

UID

Syntax

uid::=



Purpose

UID returns an integer that uniquely identifies the session user (the user who logged on).

Examples

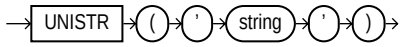
The following example returns the UID of the current user:

```
SELECT UID FROM DUAL;
```

UNISTR

Syntax

unistr::=



Purpose

`UNISTR` takes as its argument a string in any character set and returns it in Unicode in the database Unicode character set. To include UCS2 codepoint characters in the string, use the escape backslash (\) followed by the next number. To include the backslash itself, precede it with another backslash (\\).

This function is similar to the `TRANSLATE ... USING` function, except that `UNISTR` offers the escape character for UCS2 codepoints and backslash characters.

See Also:

- *Oracle9i Database Concepts* for information on Unicode character sets and character semantics
- TRANSLATE ... USING on page 6-187

Examples

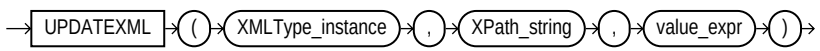
The following example returns the Unicode equivalent of its character string:

```
SELECT UNISTR('\00D6') FROM DUAL;

UN
--
Ö
```

UPDATEXML

Syntax

updatexml::=



Purpose

UPDATEXML takes as arguments an XMLType instance and an XPath-value pair, and returns an XMLType instance with the updated value. If *XPath_string* is an XML element, then the corresponding *value_expr* must be an XMLType instance. If *XPath_string* is an attribute or text node, then the *value_expr* can be any scalar datatype. The datatypes of the target of *XPath_string* and the *value_expr* must match.

If you update an XML element to null, Oracle removes the attributes and children of the element, and the element becomes empty. If you update the text node of an element to null, Oracle removes the text value of the element, and the element itself remains but is empty.

In most cases, this function materializes an XML document in memory and updates the value. However, UPDATEXML is optimized for UPDATE statements on object-relational columns so that the function updates the value directly in the column. This optimization requires the following conditions:

- The *XMLType_instance* must be the same as the column in the UPDATE ... SET clause.
- The *XPath_string* must resolve to scalar content.

Examples

The following example updates to 4 the number of docks in the San Francisco warehouse in the sample schema OE, which has a warehouse_spec column of type XMLType:

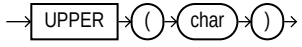
```
SELECT warehouse_name,
   EXTRACT(warehouse_spec, '/Warehouse/Docks')
   "Number of Docks"
   FROM warehouses
   WHERE warehouse_name = 'San Francisco';

WAREHOUSE_NAME       Number of Docks
-------------------- --------------------
San Francisco        <Docks>1</Docks>

UPDATE warehouses SET warehouse_spec =
   UPDATEXML(warehouse_spec,
   '/Warehouse/Docks/text()',4)
   WHERE warehouse_name = 'San Francisco';

1 row updated.
```

```
SELECT warehouse_name,
   EXTRACT(warehouse_spec, '/Warehouse/Docks')
   "Number of Docks"
   FROM warehouses
   WHERE warehouse_name = 'San Francisco';

WAREHOUSE_NAME       Number of Docks
-------------------- --------------------
San Francisco        <Docks>4</Docks>
```

UPPER

Syntax

upper::=



Purpose

UPPER returns *char*, with all letters uppercase. *char* can be any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The return value is the same datatype as *char*.

Examples

The following example returns a string in uppercase:

```
SELECT UPPER('Large') "Uppercase"
     FROM DUAL;

Upper
-----
LARGE
```

USER

Syntax

user::=

Purpose

USER returns the name of the session user (the user who logged on) with the datatype VARCHAR2. Oracle compares values of this function with blank-padded comparison semantics.

In a distributed SQL statement, the UID and USER functions identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint.

Examples

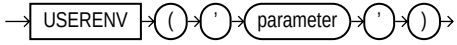
The following example returns the current user and the user's UID:

```
SELECT USER, UID FROM DUAL;
```

USERENV

Syntax

userenv::=



Purpose

> **Note:** USERENV is a legacy function that is retained for backward compatibility. Oracle Corporation recommends that you use the SYS_CONTEXT function with the built-in USERENV namespace for current functionality. See SYS_CONTEXT on page 6-153 for more information.

USERENV returns information about the current session. This information can be useful for writing an application-specific audit trail table or for determining the language-specific characters currently used by your session. You cannot use USERENV in the condition of a CHECK constraint. Table 6–3 describes the values for the *parameter* argument.

All calls to USERENV return VARCHAR2 data except for calls with the SESSIONID, ENTRYID, and COMMITSCN parameters, which return NUMBER.

Table 6–3 Parameters of the USERENV Function

Parameter	Return Value
CLIENT_INFO	CLIENT_INFO returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package. **Caution:** Some commercial applications may be using this context value. Check the applicable documentation for those applications to determine what restrictions they may impose on use of this context area. **See Also:** ■ *Oracle9i Database Concepts* for more on application context ■ CREATE CONTEXT on page 13-12 and SYS_CONTEXT on page 6-153
ENTRYID	ENTRYID returns available auditing entry identifier. You cannot use this attribute in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.
ISDBA	ISDBA returns 'TRUE' if the user has been authenticated as having DBA privileges either through the operating system or through a password file.
LANG	LANG returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	LANGUAGE returns the language and territory currently used by your session along with the database character set in this form: language_territory.characterset
SESSIONID	SESSIONID returns your auditing session identifier. You cannot use this attribute in distributed SQL statements.
TERMINAL	TERMINAL returns the operating system identifier for your current session's terminal. In distributed SQL statements, this attribute returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations.

Examples

The following example returns the LANGUAGE parameter of the current session:

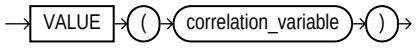
```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;
```

```
Language
----------------------------------
AMERICAN_AMERICA.WE8DEC
```

VALUE

Syntax

value::=



Purpose

VALUE takes as its argument a correlation variable (table alias) associated with a row of an object table and returns object instances stored in the object table. The type of the object instances is the same type as the object table.

Examples

The following example uses the sample table `oe.persons`, which is created in "Substitutable Table and Column Examples" on page 15-67: `SELECT VALUE(p) FROM persons p;`

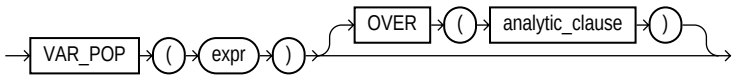
```
VALUE(P)(NAME, SSN)
-------------------------------------------------------------
PERSON_T('Bob', 1234)
EMPLOYEE_T('Joe', 32456, 12, 100000)
PART_TIME_EMP_T('Tim', 5678, 13, 1000, 20)
```

> **See Also:** "IS OF type Conditions" on page 5-19 for information on using `IS OF` type conditions with the VALUE function

VAR_POP

Syntax

var_pop::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

VAR_POP returns the population variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. If the function is applied to an empty set, then it returns null. The function makes the following calculation:

$(SUM(expr^2) - SUM(expr)^2 / COUNT(expr)) / COUNT(expr)$

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the population variance of the salaries in the employees table:

```
SELECT VAR_POP(salary) FROM employees;

VAR_POP(SALARY)
---------------
     15140307.5
```

Analytic Example

The following example calculates the cumulative population and sample variances of the monthly sales in 1998:

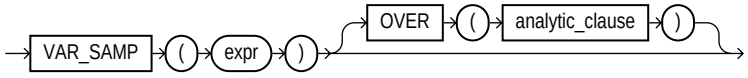
```
SELECT t.calendar_month_desc,
   VAR_POP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Pop",
   VAR_SAMP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Samp"
  FROM sales s, times t
  WHERE s.time_id = t.time_id AND t.calendar_year = 1998
  GROUP BY t.calendar_month_desc;
```

```
CALENDAR    Var_Pop   Var_Samp
-------- ---------- ----------
1998-01           0
1998-02  6.1321E+11 1.2264E+12
1998-03  4.7058E+11 7.0587E+11
1998-04  4.6929E+11 6.2572E+11
1998-05  1.5524E+12 1.9405E+12
1998-06  2.3711E+12 2.8453E+12
1998-07  3.7464E+12 4.3708E+12
1998-08  3.7852E+12 4.3260E+12
1998-09  3.5753E+12 4.0222E+12
1998-10  3.4343E+12 3.8159E+12
1998-11  3.4245E+12 3.7669E+12
1998-12  4.8937E+12 5.3386E+12
```

VAR_SAMP

Syntax

var_samp::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

VAR_SAMP returns the sample variance of a set of numbers after discarding the nulls in this set. You can use it as both an aggregate and analytic function.

The *expr* is a number expression, and the function returns a value of type NUMBER. If the function is applied to an empty set, then it returns null. The function makes the following calculation:

$$(SUM(expr^2) - SUM(expr)^2 / COUNT(expr)) / (COUNT(expr) - 1)$$

This function is similar to VARIANCE, except that given an input set of one element, VARIANCE returns 0 and VAR_SAMP returns null.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example returns the sample variance of the salaries in the sample `employees` table.

```
SELECT VAR_SAMP(salary) FROM employees;

VAR_SAMP(SALARY)
----------------
      15283140.5
```

Analytic Example

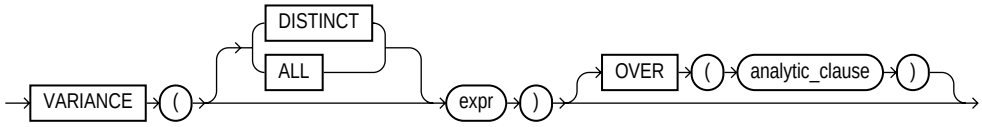
The following example calculates the cumulative population and sample variances of the monthly sales in 1998:

```
SELECT t.calendar_month_desc,
   VAR_POP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Pop",
   VAR_SAMP(SUM(s.amount_sold))
      OVER (ORDER BY t.calendar_month_desc) "Var_Samp"
  FROM sales s, times t
  WHERE s.time_id = t.time_id AND t.calendar_year = 1998
  GROUP BY t.calendar_month_desc;

CALENDAR    Var_Pop   Var_Samp
-------- ---------- ----------
1998-01           0
1998-02  6.1321E+11 1.2264E+12
1998-03  4.7058E+11 7.0587E+11
1998-04  4.6929E+11 6.2572E+11
1998-05  1.5524E+12 1.9405E+12
1998-06  2.3711E+12 2.8453E+12
1998-07  3.7464E+12 4.3708E+12
1998-08  3.7852E+12 4.3260E+12
1998-09  3.5753E+12 4.0222E+12
1998-10  3.4343E+12 3.8159E+12
1998-11  3.4245E+12 3.7669E+12
1998-12  4.8937E+12 5.3386E+12
```

VARIANCE

Syntax

variance::=



See Also: "Analytic Functions" on page 6-9 for information on syntax, semantics, and restrictions

Purpose

VARIANCE returns variance of *expr*. You can use it as an aggregate or analytic function.

Oracle calculates the variance of *expr* as follows:

- 0 if the number of rows in *expr* = 1
- VAR_SAMP if the number of rows in *expr* > 1

If you specify DISTINCT, then you can specify only the *query_partition_clause* of the *analytic_clause*. The *order_by_clause* and *windowing_clause* are not allowed.

See Also:

- "Aggregate Functions" on page 6-7
- "About SQL Expressions" on page 4-2 for information on valid forms of *expr*

Aggregate Example

The following example calculates the variance of all salaries in the sample employees table:

```
SELECT VARIANCE(salary) "Variance"
   FROM employees;

  Variance
----------
15283140.5
```

Analytic Example

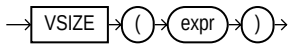
The query returns the cumulative variance of salary values in Department 30 ordered by hire date.

```
SELECT last_name, salary, VARIANCE(salary)
      OVER (ORDER BY hire_date) "Variance"
   FROM employees
   WHERE department_id = 30;

LAST_NAME           SALARY   Variance
--------------- ---------- ----------
Raphaely             11000          0
Khoo                  3100   31205000
Tobias                2800 21623333.3
Baida                 2900 16283333.3
Himuro                2600   13317000
Colmenares            2500   11307000
```

VSIZE

Syntax

vsize::=



Purpose

VSIZE returns the number of bytes in the internal representation of *expr*. If *expr* is null, then this function returns null.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion. Please refer to "Datatype Comparison Rules" on page 2-45 for more information.

Examples

The following example returns the number of bytes in the last_name of the employee in department 10:

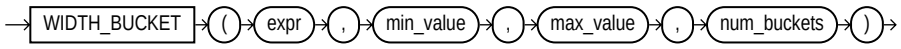
```
SELECT last_name, VSIZE (last_name) "BYTES"
```

```
  FROM employees
  WHERE department_id = 10;

LAST_NAME            BYTES
-------------- ----------
Whalen                6
```

WIDTH_BUCKET

Syntax

width_bucket::=



Purpose

`WIDTH_BUCKET` lets you construct equiwidth histograms, in which the histogram range is divided into intervals that have identical size. (Compare this function with `NTILE`, which creates equiheight histograms.) Ideally each bucket is a "closed-open" interval of the real number line. For example, a bucket can be assigned to scores between 10.00 and 19.999... to indicate that 10 is included in the interval and 20 is excluded. This is sometimes denoted [10, 20).

For a given expression, `WIDTH_BUCKET` returns the bucket number into which the value of this expression would fall after being evaluated.

- *`expr`* is the expression for which the histogram is being created. This expression must evaluate to a number or a datetime value. If *`expr`* evaluates to null, then the expression returns null.
- *`min_value`* and *`max_value`* are expressions that resolve to the end points of the acceptable range for *`expr`*. Both of these expressions must also evaluate to number or datetime values, and neither can evaluate to null.
- *`num_buckets`* is an expression that resolves to a constant indicating the number of buckets. This expression must evaluate to a positive integer.

Oracle also creates (when needed) an underflow bucket numbered 0 and an overflow bucket numbered *`num_buckets`*+1. These buckets handle values less than *`min_value`* and more than *`max_value`* and are helpful in checking the reasonableness of endpoints.

Examples

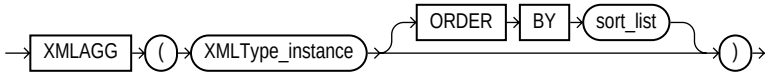
The following example creates a ten-bucket histogram on the `credit_limit` column for customers in Switzerland in the sample table `oe.customers` and returns the bucket number ("Credit Group") for each customer. Customers with credit limits greater than the maximum value are assigned to the overflow bucket, 11:

```
SELECT customer_id, cust_last_name, credit_limit,
   WIDTH_BUCKET(credit_limit, 100, 5000, 10) "Credit Group"
   FROM customers WHERE nls_territory = 'SWITZERLAND'
   ORDER BY "Credit Group";

CUSTOMER_ID CUST_LAST_NAME       CREDIT_LIMIT Credit Group
----------- -------------------- ------------ ------------
        825 Dreyfuss                      500            1
        826 Barkin                        500            1
        853 Palin                         400            1
        827 Siegel                        500            1
        843 Oates                         700            2
        844 Julius                        700            2
        835 Eastwood                     1200            3
        840 Elliott                      1400            3
        842 Stern                        1400            3
        841 Boyer                        1400            3
        837 Stanton                      1200            3
        836 Berenger                     1200            3
        848 Olmos                        1800            4
        849 Kaurusmdki                   1800            4
        828 Minnelli                     2300            5
        829 Hunter                       2300            5
        852 Tanner                       2300            5
        851 Brown                        2300            5
        850 Finney                       2300            5
        830 Dutt                         3500            7
        831 Bel Geddes                   3500            7
        832 Spacek                       3500            7
        838 Nicholson                    3500            7
        839 Johnson                      3500            7
        833 Moranis                      3500            7
        834 Idle                         3500            7
        845 Fawcett                      5000           11
        846 Brando                       5000           11
        847 Streep                       5000           11
```

XMLAGG

Syntax

XMLAgg::=



Purpose

XMLAgg is an aggregate function. It takes a collection of XML fragments and returns an aggregated XML document. Any arguments that return null are dropped from the result.

XMLAgg is similar to SYS_XMLAgg except that XMLAgg returns a collection of nodes, but it does not accept formatting using the XMLFormat object. Also, XMLAgg does not enclose the output in an element tag as does SYS_XMLAgg.

> **See Also:** XMLELEMENT on page 6-211 and SYS_XMLAGG on page 6-162

Examples

The following example produces a Department element containing Employee elements with employee job ID and last name as the contents of the elements:

```
SELECT XMLELEMENT("Department",
   XMLAGG(XMLELEMENT("Employee",
      e.job_id||' '||e.last_name)))
   AS "Dept_list"
   FROM employees e
   WHERE e.department_id = 30;

Dept_list
----------------------------------------------------
<Department><Employee>PU_MAN Raphaely</Employee>
<Employee>PU_CLERK Khoo</Employee>
<Employee>PU_CLERK Baida</Employee>
<Employee>PU_CLERK Tobias</Employee>
<Employee>PU_CLERK Himuro</Employee>
<Employee>PU_CLERK Colmenares</Employee>
</Department>

1 row selected.
```

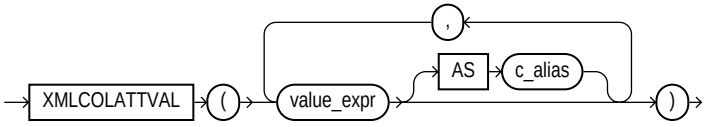
The result is a single row, because XMLAgg aggregates the rows. You can use the GROUP BY clause to group the returned set of rows into multiple groups:

```
SELECT XMLELEMENT("Department",
      XMLAGG(XMLELEMENT("Employee", e.job_id||' '||e.last_name)))
   AS "Dept_list"
   FROM employees e
   GROUP BY e.department_id;

Dept_list
----------------------------------------------------------
<Department>
  <Employee>AD_ASST Whalen</Employee>
</Department>

<Department>
  <Employee>MK_MAN Hartstein</Employee>
  <Employee>MK_REP Fay</Employee>
</Department>

<Department>
  <Employee>PU_MAN Raphaely</Employee>
  <Employee>PU_CLERK Khoo</Employee>
  <Employee>PU_CLERK Tobias</Employee>
  <Employee>PU_CLERK Baida</Employee>
  <Employee>PU_CLERK Colmenares</Employee>
  <Employee>PU_CLERK Himuro</Employee>
</Department>
...
```

XMLCOLATTVAL

Syntax

XMLColAttVal::=



Purpose

`XMLColAttVal` creates an XML fragment and then expands the resulting XML so that each XML fragment has the name "column" with the attribute "name". You can use the AS *c_alias* clause to change the value of the `name` attribute to something other than the column name.

You must specify a value for *value_expr*. If *value_expr* is null, then no element is returned.

Restriction: You cannot specify an object type column for *value_expr*.

Examples

The following example creates an `Emp` element for a subset of employees, with nested `employee_id`, `last_name`, and `salary` elements as the contents of `Emp`. Each nested element is named `column` and has a `name` attribute with the column name as the attribute value:

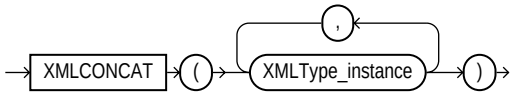
```
SELECT XMLELEMENT("Emp",
   XMLCOLATTVAL(e.employee_id, e.last_name, e.salary)) "Emp Element"
   FROM employees e
   WHERE employee_id = 204;

Emp Element
--------------------------------------------------------------------
<Emp>
  <column name="EMPLOYEE_ID">204</column>
  <column name="LAST_NAME">Baer</column>
  <column name="SALARY">10000</column>
</Emp>
```

See Also: the example for XMLFOREST on page 6-214 to compare the output of these two functions

XMLCONCAT

Syntax

XMLConcat::=



Purpose

`XMLConcat` takes as input a series of `XMLType` instances, concatenates the series of elements for each row, and returns the concatenated series. `XMLConcat` is the inverse of `XMLSequence`.

Null expressions are dropped from the result. If all the value expressions are null, then the function returns null.

See Also: XMLSEQUENCE on page 6-215

Examples

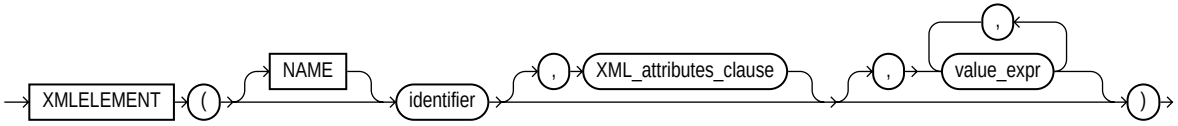
The following example creates XML elements for the first and last names of a subset of employees, and then concatenates and returns those elements:

```
SELECT XMLCONCAT(XMLELEMENT("First", e.first_name),
   XMLELEMENT("Last", e.last_name)) AS "Result"
   FROM employees e
   WHERE e.employee_id > 202;

Result
--------------------------------------------------------------------
<First>Susan</First>
<Last>Mavris</Last>

<First>Hermann</First>
<Last>Baer</Last>

<First>Shelley</First>
<Last>Higgins</Last>

<First>William</First>
<Last>Gietz</Last>

4 rows selected.
```

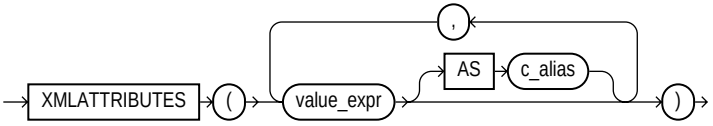
XMLELEMENT

Syntax

XMLElement::=



XML_attributes_clause::=



Purpose

`XMLElement` takes an element name for *`identifier`*, an optional collection of attributes for the element, and arguments that make up the element's content. It returns an instance of type `XMLType`. `XMLElement` is similar to `SYS_XMLGen` except that `XMLElement` can include attributes in the XML returned, but it does not accept formatting using the `XMLFormat` object.

The `XMLElement` function is typically nested to produce an XML document with a nested structure, as in the example in the following section.

You must specify a value for *`identifier`*, which Oracle uses as the enclosing tag. The identifier does not have to be a column name or column reference. It cannot be an expression or null.

In the *`XML_attributes_clause`*, if the *`value_expr`* is null, then no attribute is created for that value expression. The type of *`value_expr`* cannot be an object type or collection.

The objects that make up the element content follow the `XMLATTRIBUTES` keyword.

- If *`value_expr`* is a scalar expression, then you can omit the `AS` clause, and Oracle uses the column name as the element name.
- If *`value_expr`* is an object type or collection, then the `AS` clause is mandatory, and Oracle uses the specified *`c_alias`* as the enclosing tag.

- If *value_expr* is null, then no element is created for that value expression.

See Also: SYS_XMLGEN on page 6-163

Examples

The following example produces an Emp element for a series of employees, with nested elements that provide the employee's name and hire date:

```
SELECT XMLELEMENT("Emp", XMLELEMENT("Name",
   e.job_id||' '||e.last_name),
   XMLELEMENT("Hiredate", e.hire_date)) as "Result"
   FROM employees e
   WHERE employee_id > 200;

Result
--------------------------------------------------------------------
<Emp>
  <Name>MK_MAN Hartstein</Name>
  <Hiredate>17-FEB-96</Hiredate>
</Emp>

<Emp>
  <Name>MK_REP Fay</Name>
  <Hiredate>17-AUG-97</Hiredate>
</Emp>

<Emp>
  <Name>HR_REP Mavris</Name>
  <Hiredate>07-JUN-94</Hiredate>
</Emp>

<Emp>
  <Name>PR_REP Baer</Name>
  <Hiredate>07-JUN-94</Hiredate>
</Emp>

<Emp>
  <Name>AC_MGR Higgins</Name>
  <Hiredate>07-JUN-94</Hiredate>
</Emp>

<Emp>
  <Name>AC_ACCOUNT Gietz</Name>
  <Hiredate>07-JUN-94</Hiredate>
</Emp>
```

```
6 rows selected.
```

The following similar example uses the XMLElement function with the *XML_attributes_clause* to create nested XML elements with attribute values for the top-level element:

```
SELECT XMLELEMENT("Emp",
      XMLATTRIBUTES(e.employee_id AS "ID", e.last_name),
      XMLELEMENT("Dept", e.department_id),
      XMLELEMENT("Salary", e.salary)) AS "Emp Element"
   FROM employees e
   WHERE e.employee_id = 206;

Emp Element
--------------------------------------------------------------
<Emp ID="206" LAST_NAME="Gietz">
  <Dept>110</Dept>
  <Salary>8300</Salary>
</Emp>
```

Notice that the AS *identifier* clause was not specified for the last_name column. As a result, the XML returned uses the column name last_name as the default.

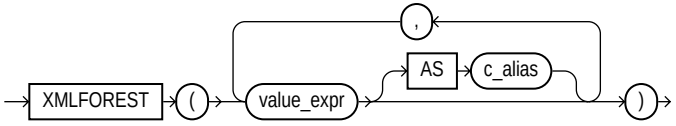
Finally, the next example uses a subquery within the *XML_attributes_clause* to retrieve information from another table into the attributes of an element:

```
SELECT XMLELEMENT("Emp", XMLATTRIBUTES(e.employee_id, e.last_name),
   XMLELEMENT("Dept", XMLATTRIBUTES(e.department_id,
   (SELECT d.department_name FROM departments d
   WHERE d.department_id = e.department_id) as "Dept_name")),
   XMLELEMENT("salary", e.salary),
   XMLELEMENT("Hiredate", e.hire_date)) AS "Emp Element"
   FROM employees e
   WHERE employee_id = 205;

Emp Element
-----------------------------------------------------------------
<Emp EMPLOYEE_ID="205" LAST_NAME="Higgins">
  <Dept DEPARTMENT_ID="110" Dept_name="Accounting"/>
  <salary>12000</salary>
  <Hiredate>07-JUN-94</Hiredate>
</Emp>
```

XMLFOREST

Syntax

XMLForest::=



Purpose

`XMLForest` converts each of its argument parameters to XML, and then returns an XML fragment that is the concatenation of these converted arguments.

- If *`value_expr`* is a scalar expression, then you can omit the AS clause, and Oracle uses the column name as the element name.
- If *`value_expr`* is an object type or collection, then the AS clause is mandatory, and Oracle uses the specified *`c_alias`* as the enclosing tag.
- If *`value_expr`* is null, then no element is created for that *`value_expr`*.

Examples

The following example creates an `Emp` element for a subset of employees, with nested `employee_id`, `last_name`, and `salary` elements as the contents of `Emp`:

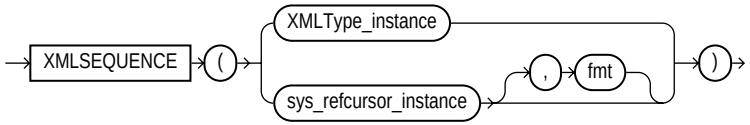
```
SELECT XMLELEMENT("Emp",
   XMLFOREST(e.employee_id, e.last_name, e.salary))
   "Emp Element"
   FROM employees e
   WHERE employee_id = 204;

Emp Element
--------------------------------------------------------------------
<Emp>
  <EMPLOYEE_ID>204</EMPLOYEE_ID>
  <LAST_NAME>Baer</LAST_NAME>
  <SALARY>10000</SALARY>
</Emp>
```

See Also: the example for XMLCOLATTVAL on page 6-208 to compare the output of these two functions

XMLSEQUENCE

Syntax

XMLSequence::=



Purpose

XMLSequence has two forms:

- The first form takes as input an XMLType instance and returns a varray of the top-level nodes in the XMLType.
- The second form takes as input a REFCURSOR instance, with an optional instance of the XMLFormat object, and returns as an XMLSequence type an XML document for each row of the cursor.

Because XMLSequence returns a collection of XMLType, you can use this function in a TABLE clause to unnest the collection values into multiple rows, which can in turn be further processed in the SQL query.

> **See Also:** *Oracle9i XML API Reference - XDK and Oracle XML DB* for more information on this function

Examples

The following example shows how XMLSequence divides up an XML document with multiple elements into VARRAY single-element documents. In this example, the TABLE keyword instructs Oracle to consider the collection a table value that can be used in the FROM clause of the subquery:

```
SELECT EXTRACT(warehouse_spec, '/Warehouse') as "Warehouse"
   FROM warehouses WHERE warehouse_name = 'San Francisco';

Warehouse
------------------------------------------------------------
<Warehouse>
  <Building>Rented</Building>
  <Area>50000</Area>
  <Docks>1</Docks>
  <DockType>Side load</DockType>
```

```
  <WaterAccess>Y</WaterAccess>
  <RailAccess>N</RailAccess>
  <Parking>Lot</Parking>
  <VClearance>12 ft</VClearance>
</Warehouse>

1 row selected.

SELECT VALUE(p)
   FROM warehouses w,
   TABLE(XMLSEQUENCE(EXTRACT(warehouse_spec, '/Warehouse/*'))) p
   WHERE w.warehouse_name = 'San Francisco';

VALUE(P)
--------------------------------------------------------------------
<Building>Rented</Building>
<Area>50000</Area>
<Docks>1</Docks>
<DockType>Side load</DockType>
<WaterAccess>Y</WaterAccess>
<RailAccess>N</RailAccess>
<Parking>Lot</Parking>
<VClearance>12 ft</VClearance>

8 rows selected.
```

XMLTRANSFORM

Syntax

XMLTransform::=

Purpose

`XMLTransform` takes as arguments an `XMLType` instance and an XSL style sheet, which is itself a form of `XMLType` instance. It applies the style sheet to the instance and returns an `XMLType`.

This function is useful for organizing data according to a style sheet as you are retrieving it from the database.

See Also: *Oracle9i XML API Reference - XDK and Oracle XML DB* for more information on this function

Examples

The `XMLTransform` function requires the existence of an XSL style sheet. Here is an example of a very simple style sheet that alphabetizes elements within a node:

```
CREATE TABLE xsl_tab (col1 XMLTYPE);

INSERT INTO xsl_tab VALUES (
   XMLTYPE.createxml(
   '<?xml version="1.0"?>
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
      <xsl:output encoding="utf-8"/>
      <!-- alphabetizes an xml tree -->
      <xsl:template match="*">
        <xsl:copy>
          <xsl:apply-templates select="*|text()">
            <xsl:sort select="name(.)" data-type="text" order="ascending"/>
          </xsl:apply-templates>
        </xsl:copy>
      </xsl:template>
      <xsl:template match="text()">
        <xsl:value-of select="normalize-space(.)"/>
      </xsl:template>
    </xsl:stylesheet>  '));

1 row created.
```

The next example uses the `xsl_tab` XSL style sheet to alphabetize the elements in one `warehouse_spec` of the sample table `oe.warehouses`:

```
SELECT XMLTRANSFORM(w.warehouse_spec, x.col1).GetClobVal()
   FROM warehouses w, xsl_tab x
   WHERE w.warehouse_name = 'San Francisco';

XMLTRANSFORM(W.WAREHOUSE_SPEC,X.COL1).GETCLOBVAL()
--------------------------------------------------------------------------------
<Warehouse>
  <Area>50000</Area>
  <Building>Rented</Building>
  <DockType>Side load</DockType>
  <Docks>1</Docks>
  <Parking>Lot</Parking>
```

```
  <RailAccess>N</RailAccess>
  <VClearance>12 ft</VClearance>
  <WaterAccess>Y</WaterAccess>
</Warehouse>
```

ROUND and TRUNC Date Functions

Table 6–4 lists the format models you can use with the ROUND and TRUNC date functions and the units to which they round and truncate dates. The default model, 'DD', returns the date rounded or truncated to the day with a time of midnight.

Table 6–4 Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
CC SCC	One greater than the first two digits of a four-digit year
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	ISO Year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year

Table 6–4 (Cont.) Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter NLS_TERRITORY.

> **See Also:** *Oracle9i Database Reference* and *Oracle9i Database Globalization Support Guide* for information on this parameter

User-Defined Functions

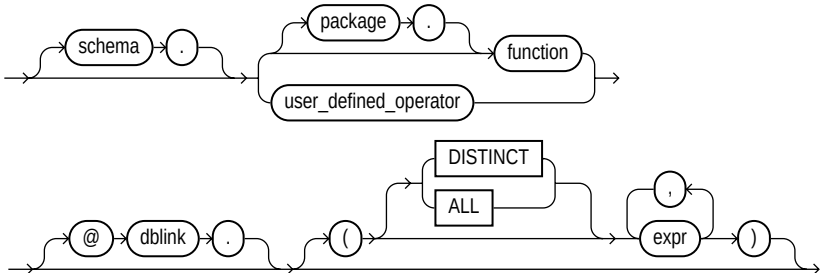
You can write user-defined functions in PL/SQL or Java to provide functionality that is not available in SQL or SQL built-in functions. User-defined functions can appear in a SQL statement anywhere SQL functions can appear, that is, wherever an expression can occur.

For example, user-defined functions can be used in the following:

- The select list of a SELECT statement
- The condition of a WHERE clause
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

Note: Oracle SQL does not support calling of functions with boolean parameters or returns. Therefore, if your user-defined functions will be called from SQL statements, you must design them to return numbers (0 or 1) or character strings ('TRUE' or 'FALSE').

user_defined_function::=



The optional expression list must match attributes of the function, package, or operator.

Restriction: The DISTINCT and ALL keywords are valid only with a user-defined aggregate function.

See Also:

- CREATE FUNCTION on page 13-49 for information on creating functions, including restrictions on user-defined functions
- *Oracle9i Application Developer's Guide - Fundamentals* for a complete description on the creation and use of user functions

Prerequisites

User-defined functions must be created as top-level functions or declared with a package specification before they can be named within a SQL statement.

To use a user function in a SQL expression, you must own or have EXECUTE privilege on the user function. To query a view defined with a user function, you must have SELECT privileges on the view. No separate EXECUTE privileges are needed to select from the view.

See Also:

- CREATE FUNCTION on page 13-49 for information on creating top-level functions
- CREATE PACKAGE on page 14-50 for information on specifying packaged functions

Name Precedence

Within a SQL statement, the names of database columns take precedence over the names of functions with no parameters. For example, if the Human Resources manager creates the following two objects in the hr schema:

```
CREATE TABLE new_emps (new_sal NUMBER, ...);
CREATE FUNCTION new_sal RETURN NUMBER IS BEGIN ... END;
```

then in the following two statements, the reference to new_sal refers to the column new_emps.new_sal:

```
SELECT new_sal FROM new_emps;
SELECT new_emps.new_sal FROM new_emps;
```

To access the function new_sal, you would enter:

```
SELECT hr.new_sal FROM new_emps;
```

Here are some sample calls to user functions that are allowed in SQL expressions:

```
circle_area (radius)
payroll.tax_rate (empno)
scott.payroll.tax_rate (dependent, empno)@ny
```

Example To call the tax_rate user function from schema hr, execute it against the ss_no and sal columns in tax_table, and place the results in the variable income_tax, specify the following:

```
SELECT hr.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table
    WHERE ss_no = tax_id;
```

Naming Conventions

If only one of the optional schema or package names is given, then the first identifier can be either a schema name or a package name. For example, to

determine whether PAYROLL in the reference PAYROLL.TAX_RATE is a schema or package name, Oracle proceeds as follows:

1. Check for the PAYROLL package in the current schema.
2. If a PAYROLL package is not found, then look for a schema name PAYROLL that contains a top-level TAX_RATE function. If no such function is found, then return an error.
3. If the PAYROLL package is found in the current schema, then look for a TAX_RATE function in the PAYROLL package. If no such function is found, then return an error.

You can also refer to a stored top-level function using any synonym that you have defined for it.

7

Common SQL DDL Clauses

This chapter describes some SQL data definition clauses that appear in multiple SQL statements. This chapter contains these sections:

allocate_extent_clause

Purpose

Use the *allocate_extent_clause* clause to explicitly allocate a new extent for a database object.

Explicitly allocating an extent with this clause does not change the values of the NEXT and PCTINCREASE storage parameters, so does not affect the size of the next extent to be allocated implicitly by Oracle.

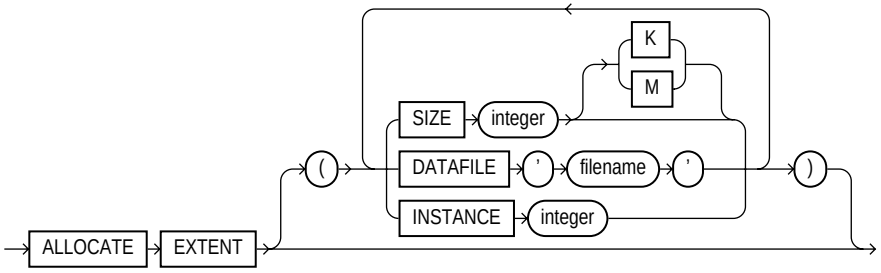
> **See Also:** *storage_clause* on page 7-56 for information about the NEXT and PCTINCREASE storage parameters

You can allocate an extent in the following SQL statements:

- ALTER CLUSTER (see ALTER CLUSTER on page 9-7)
- ALTER INDEX: to allocate an extent to the index, an index partition, or an index subpartition (see ALTER INDEX on page 9-64)
- ALTER MATERIALIZED VIEW: to allocate an extent to the materialized view, one of its partitions or subpartitions, or the overflow segment of an index-organized materialized view (see ALTER MATERIALIZED VIEW on page 9-7)
- ALTER MATERIALIZED VIEW LOG (see ALTER MATERIALIZED VIEW LOG on page 9-112)
- ALTER TABLE: to allocate an extent to the table, a table partition, a table subpartition, the mapping table of an index-organized table, the overflow segment of an index-organized table, or a LOB storage segment (see ALTER TABLE on page 11-2)

Syntax

allocate_extent_clause::=



Keywords and Parameters

This section describes the parameters of the *allocate_extent_clause*. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

Note: You cannot specify the *allocate_extent_clause* and the *deallocate_unused_clause* in the same statement.

SIZE

Specify the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes.

For a table, index, materialized view, or materialized view log, if you omit SIZE, then Oracle determines the size based on the values of the storage parameters of the object. However, for a cluster, Oracle does not evaluate the cluster's storage parameters, so you must specify SIZE if you do not want Oracle to use a default value.

DATAFILE '*filename*'

Specify one of the datafiles in the tablespace of the table, cluster, index, materialized view, or materialized view log to contain the new extent. If you omit DATAFILE, then Oracle chooses the datafile.

INSTANCE *integer*

Use this parameter only if you are using Oracle with Real Application Clusters.

Specifying INSTANCE *integer* makes the new extent available to the freelist group associated with the specified instance. If the instance number exceeds the maximum number of freelist groups, then Oracle divides the specified number by the maximum number and uses the remainder to identify the freelist group to be used. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER.

If you omit this parameter, then the space is allocated to the table, cluster, index, materialized view, or materialized view log but is not drawn from any particular freelist group. Instead, Oracle uses the master freelist and allocates space as needed.

Note: If you are using automatic segment-space management, then the INSTANCE parameter of the *allocate_extent_clause* may not reserve the newly allocated space for the specified instance, because automatic segment-space management does not maintain rigid affinity between extents and instances.

See Also: *Oracle9i Real Application Clusters Administration* for more information on setting the INSTANCE parameter of *allocate_extent_clause*

constraints

Purpose

Use one of the *constraints* to define an **integrity constraint—**a rule that restricts the values in a database. Oracle lets you create six types of constraints and lets you declare them in two ways.

The six types of integrity constraint are described briefly here and more fully in "Keywords and Parameters" on page 7-10:

- A **NOT NULL constraint** prohibits a database value from being null.
- A **unique constraint** prohibits multiple rows from having the same value in the same column or combination of columns but allows some values to be null.
- A **primary key constraint** combines a NOT NULL constraint and a unique constraint in a single declaration. That is, it prohibits multiple rows from having the same value in the same column or combination of columns and prohibits values from being null.
- A **foreign key constraint** requires values in one table to match values in another table.
- A **check constraint** requires a value in the database to comply with a specified condition.
- A REF column by definition references an object in another object type or in a relational table. A **REF constraint** lets you further describe the relationship between the REF column and the object it references.

You can define constraints syntactically in two ways:

- As part of the definition of an individual column or attribute. This is called **inline** specification.
- As part of the table definition. This is called **out-of-line** specification.

NOT NULL constraints must be declared inline. All other constraints can be declared either inline or out of line.

Constraint clauses can appear in the following statements:

- CREATE TABLE (see CREATE TABLE on page 15-7)
- ALTER TABLE (see ALTER TABLE on page 11-2)
- CREATE VIEW (see CREATE VIEW on page 16-39)

- ALTER VIEW (see ALTER VIEW on page 12-30)

View Constraints Oracle does not enforce view constraints. However, you can enforce constraints on views through constraints on base tables.

You can specify only unique, primary key, and foreign key constraints on views, and they are supported only in DISABLE NOVALIDATE mode. You cannot define view constraints on attributes of an object column.

See Also:

- "View Constraints" on page 7-26 for additional information on view constraints
- "DISABLE Clause" on page 7-21 for information on DISABLE NOVALIDATE mode

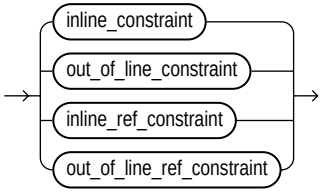
Prerequisites

You must have the privileges necessary to issue the statement in which you are defining the constraint.

To create a foreign key constraint, in addition, the parent table or view must be in your own schema, or you must have the REFERENCES privilege on the columns of the referenced key in the parent table or view.

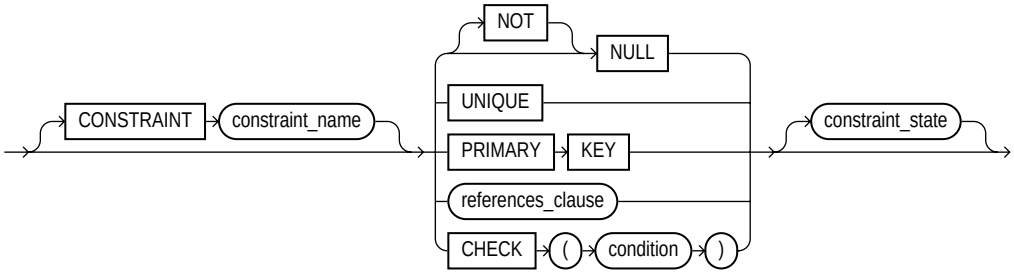
Syntax

constraints::=



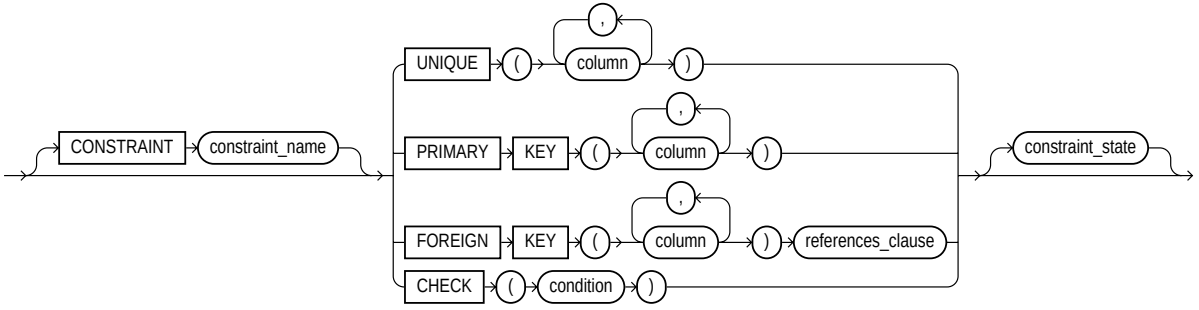
(*inline_constraint::=* on page 7-7, *out_of_line_constraint::=* on page 7-7, *inline_ref_constraint::=* on page 7-7, *out_of_line_ref_constraint::=* on page 7-8)

inline_constraint::=



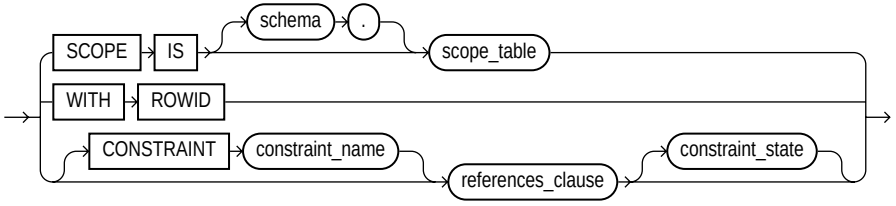
(*references_clause::=* on page 7-8)

out_of_line_constraint::=



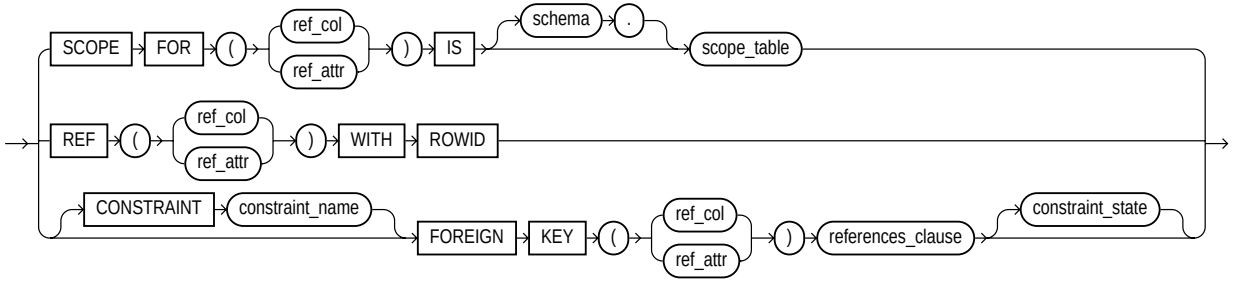
(*references_clause::=* on page 7-8, *constraint_state::=* on page 7-8)

inline_ref_constraint::=



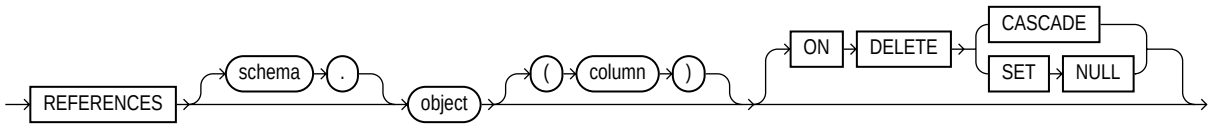
(*references_clause::=* on page 7-8, *constraint_state::=* on page 7-8)

out_of_line_ref_constraint::=

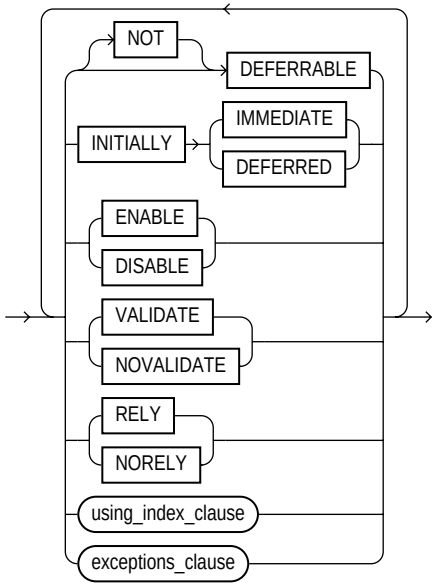


(*references_clause::=* on page 7-8, *constraint_state::=* on page 7-8)

references_clause::=

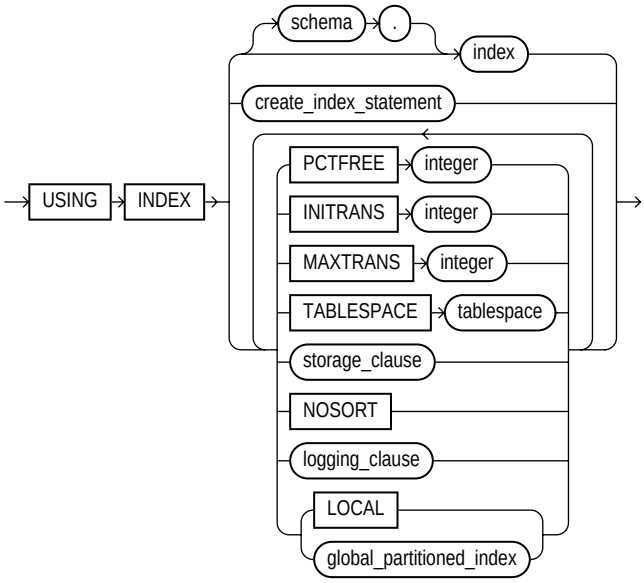


constraint_state::=



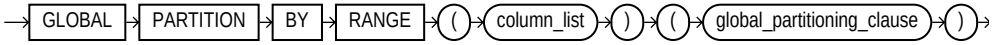
(*using_index_clause::=* on page 7-9, *exceptions_clause::=* on page 7-10)

using_index_clause::=

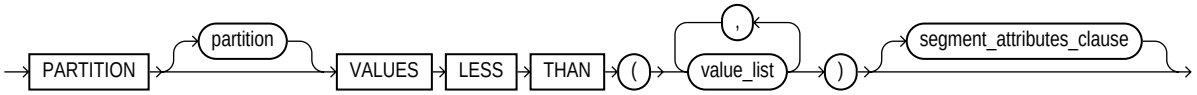


(*create_index_statement*: *create_index::=* on page 13-63, *storage_clause*s on page 7-56, *logging_clause* on page 7-45, *global_partitioned_index::=* on page 7-9)

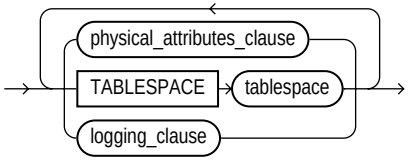
global_partitioned_index::=



index_partitioning_clause::=

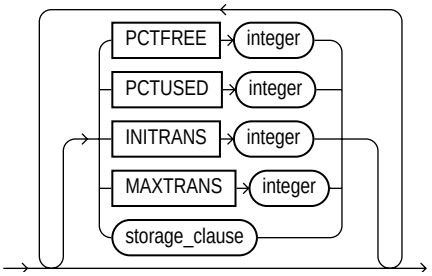


segment_attributes_clause::=



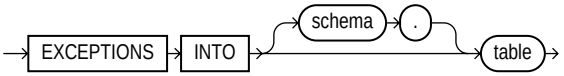
(*physical_attributes_clause::=* on page 7-53, *logging_clause* on page 7-45)

physical_attributes_clause::=



(*storage_clause* on page 7-56)

exceptions_clause::=



Keywords and Parameters

This section describes the semantics of the *constraints*. For additional information, refer to the SQL statement in which you define or redefine a constraint for a table or view.

Note: Oracle does not support constraints on columns or attributes whose type is a user-defined object, nested table, VARRAY, REF, or LOB, with two exceptions:

- NOT NULL constraints are supported for a column or attribute whose type is user-defined object, VARRAY, REF, or LOB.
- NOT NULL, foreign key, and REF constraints are supported on a column of type REF.

CONSTRAINT *constraint_name* Specify a name for the constraint. If you omit this identifier, then Oracle generates a name with the form SYS_C*n*. Oracle stores the name and the definition of the integrity constraint in the USER_, ALL_, and DBA_ CONSTRAINTS data dictionary views (in the CONSTRAINT_NAME and SEARCH_ CONDITION columns, respectively).

See Also: *Oracle9i Database Reference* for information on the data dictionary views

NOT NULL Constraints

A NOT NULL constraint prohibits a column from containing nulls. The NULL keyword does not actually define an integrity constraint, but you can specify it to explicitly permit a column to contain nulls. You must define NOT NULL and NULL using inline specification. If you specify neither NOT NULL nor NULL, then the default is NULL.

NOT NULL constraints are the only constraints you can specify inline on XMLType and VARRAY columns.

To satisfy a NOT NULL constraint, every row in the table must contain a value for the column.

Note: Oracle does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, then you must either specify NOT NULL constraints for at least one of the index key columns or create a bitmap index.

Restrictions on NOT NULL Constraints:

- You cannot specify NULL or NOT NULL in a view constraint.
- You cannot specify NULL or NOT NULL for an attribute of an object. Instead, use a CHECK constraint with the IS [NOT] NULL condition.

See Also: "Attribute-Level Constraints Example" on page 7-34 and "NOT NULL Example" on page 7-29

Unique Constraints

A **unique** constraint designates a column as a unique key. A **composite unique key** designates a combination of columns as the unique key. When you define a unique constraint inline, you need only the UNIQUE keyword. When you define a unique constraint out of line, you must also specify one or more columns. You must define a composite unique key out of line.

To satisfy a unique constraint, no two rows in the table can have the same value for the unique key. However, the unique key made up of a single column can contain

nulls. To satisfy a composite unique key, no two rows in the table or view can have the same combination of values in the key columns. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.

Note: When you specify a unique constraint on one or more columns, Oracle implicitly creates an index on the unique key. If you are defining uniqueness for purposes of query performance, then Oracle Corporation recommends that you instead create the unique index explicitly using a `CREATE UNIQUE INDEX` statement. See CREATE INDEX on page 13-62 for more information.

Restrictions on Unique Constraints:

- A table or view can have only one unique key.
- None of the columns in the unique key can have datatype LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `OBJECT`, `BFILE`, or `REF`, or `TIMESTAMP WITH TIME ZONE`. However, the unique key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- A composite unique key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.
- You cannot specify a unique key when creating a subtable or a subview in an inheritance hierarchy. The unique key can be specified only for the top-level (root) table or view.

See Also: "Unique Key Example" on page 7-27 and Composite Unique Key Example on page 7-27

Primary Key Constraints

A **primary key** constraint designates a column as the primary key of a table or view. A **composite primary key** designates a combination of columns as the primary key. When you define a primary key constraint inline, you need only the `PRIMARY KEY` keywords. When you define a primary key constraint out of line, you must also specify one or more columns. You must define a composite primary key out of line.

A primary key constraint combines a `NOT NULL` and unique constraint in one declaration. Therefore, to satisfy a primary key constraint:

- No primary key value can appear in more than one row in the table.
- No column that is part of the primary key can contain a null.

Restrictions on Primary Key Constraints:

- A table or view can have only one primary key.
- None of the columns in the primary key can have datatype LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `OBJECT`, `BFILE`, or `REF`, or `TIMESTAMP WITH TIME ZONE`. However, the primary key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- The size of the primary key cannot exceed approximately one database block.
- A composite primary key cannot have more than 32 columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.
- You cannot specify a primary key when creating a subtable or a subview in an inheritance hierarchy. The primary key can be specified only for the top-level (root) table or view.

See Also: "Primary Key Example" on page 7-28 and "Composite Primary Key Example" on page 7-29

Foreign Key Constraints

A **foreign key constraint** (also called a **referential integrity constraint**) designates a column as the foreign key and establishes a relationship between that foreign key and a specified primary or unique key, called the **referenced key**. A **composite foreign key** designates a combination of columns as the foreign key.

The table or view containing the foreign key is called the **child** object, and the table or view containing the referenced key is called the **parent** object. The foreign key and the referenced key can be in the same table or view. In this case, the parent and child tables are the same. If you identify only the parent table or view and omit the column name, then the foreign key automatically references the primary key of the parent table or view. The corresponding column or columns of the foreign key and the referenced key must match in order and datatype.

You can define a foreign key constraint on a single key column either inline or out of line. You must specify a composite foreign key and a foreign key on an attribute out of line.

To satisfy a composite foreign key constraint, the composite foreign key must refer to a composite unique key or a composite primary key in the parent table or view, or the value of at least one of the columns of the foreign key must be null.

You can designate the same column or combination of columns as both a foreign key and a primary or unique key. You can also designate the same column or combination of columns as both a foreign key and a cluster key.

You can define multiple foreign keys in a table or view. Also, a single column can be part of more than one foreign key.

Restrictions on Foreign Key Constraints:

- None of the columns in the foreign key can have datatype LOB, `LONG`, `LONG RAW`, `VARRAY`, `NESTED TABLE`, `OBJECT`, `BFILE`, or `REF`, or `TIMESTAMP WITH TIME ZONE`. However, the primary key can contain a column of `TIMESTAMP WITH LOCAL TIME ZONE`.
- The referenced unique or primary key constraint on the parent table or view must already be defined.
- A composite foreign key cannot have more than 32 columns.
- The child and parent tables must be on the same database. To enable referential integrity constraints across nodes of a distributed database, you must use database triggers. See CREATE TRIGGER on page 15-95.
- If either the child or parent object is a view, then the constraint is subject to all restrictions on view constraints. See "View Constraints" on page 7-26.
- You cannot define a foreign key constraint in a `CREATE TABLE` statement that contains an `AS` *`subquery`* clause. Instead, you must create the table without the constraint and then add it later with an `ALTER TABLE` statement.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* and "Foreign Key Constraint Example" on page 7-29 and "Composite Foreign Key Constraint Example" on page 7-31

references_clause Foreign key constraints use the *`references_clause`* syntax. When you specify a foreign key constraint inline, you need only the *`references_clause`*. When you specify a foreign key constraint out of line, you must also specify the `FOREIGN KEY` keywords and one or more columns.

ON DELETE Clause The `ON DELETE` clause lets you determine how Oracle automatically maintains referential integrity if you remove a referenced primary or unique key value. If you omit this clause, then Oracle does not allow you to delete

referenced key values in the parent table that have dependent rows in the child table.

- Specify CASCADE if you want Oracle to remove dependent foreign key values.
- Specify SET NULL if you want Oracle to convert dependent foreign key values to NULL.

Restriction on the ON DELETE Clause: You cannot specify this clause for a view constraint.

> **See Also:** "ON DELETE Example" on page 7-30

Check Constraints

A **check constraint** lets you specify a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null). When Oracle evaluates a check constraint condition for a particular row, any column names in the condition refer to the column values in that row.

The syntax for inline and out-of-line specification of check constraints is the same. However, inline specification can refer only to the column (or the attributes of the column if it is an object column) currently being defined, whereas out-of-line specification can refer to multiple columns or attributes.

Note: Oracle does not verify that conditions of check constraints are not mutually exclusive. Therefore, if you create multiple check constraints for a column, design them carefully so their purposes do not conflict. Do not assume any particular order of evaluation of the conditions.

> **See Also:**
>
> - Chapter 5, "Conditions" for additional information and syntax
> - "Check Constraint Examples" on page 7-31 and "Attribute-Level Constraints Example" on page 7-34

Restrictions on Check Constraints:

- You cannot specify a check constraint for a view. However, you can define the view using the WITH CHECK OPTION clause, which is equivalent to specifying a check constraint for the view.

- The condition of a check constraint can refer to any column in the table, but it cannot refer to columns of other tables.
- Conditions of check constraints cannot contain the following constructs:
  - Subqueries and scalar subquery expressions
  - Calls to the functions that are not deterministic (CURRENT_DATE, CURRENT_TIMESTAMP, DBTIMEZONE, LOCALTIMESTAMP, SESSIONTIMEZONE, SYSDATE, SYSTIMESTAMP, UID, USER, and USERENV)
  - Calls to user-defined functions
  - Dereferencing of REF columns (for example, using the DEREF function)
  - Nested table columns or attributes
  - The pseudocolumns CURRVAL, NEXTVAL, LEVEL, or ROWNUM
  - Date constants that are not fully specified

REF Constraints

REF constraints let you describe the relationship between a column of type REF and the object it references.

ref_constraint REF constraints use the *ref_constraint* syntax. You define a REF constraint either inline or out of line. Out-of-line specification requires you to specify the REF column or attribute you are further describing.

- For *ref_column*, specify the name of a REF column of an object or relational table.
- For *ref_attribute*, specify an embedded REF attribute within an object column of a relational table.

Both inline and out-of-line specification let you define a scope constraint, a rowid constraint, or a referential integrity constraint on a REF column.

If the REF column's scope table or referenced table has a primary-key-based object identifier, then it is a **user-defined REF column**.

> **See Also:** *Oracle9i Database Concepts* for more information on REFs, "Foreign Key Constraints" on page 7-13, and "REF Constraint Examples" on page 7-34

SCOPE REF Constraints

In a table with a REF column, each REF value in the column can conceivably reference a row in a different object table. The SCOPE clause restricts the scope of references to a single table, *scope_table*. The values in the REF column or attribute point to objects in *scope_table*, in which object instances (of the same type as the REF column) are stored.

Specify the SCOPE clause to restrict the scope of references in the REF column to a single table. For you to specify this clause, *scope_table* must be in your own schema or you must have SELECT privileges on *scope_table* or SELECT ANY TABLE system privileges. You can specify only one scope table for each REF column.

Restrictions on Scope Constraints:

- You cannot add a scope constraint to an existing column unless the table is empty.
- You cannot specify a scope constraint for the REF elements of a VARRAY column.
- You must specify this clause if you specify AS *subquery* and the subquery returns user-defined REFs.
- You cannot subsequently drop a scope constraint from a REF column.

Rowid REF Constraints

Specify WITH ROWID to store the rowid along with the REF value in *ref_column* or *ref_attribute*. Storing the rowid with the REF value can improve the performance of dereferencing operations, but will also use more space. Default storage of REF values is without rowids.

> **See Also:** the function DEREF on page 6-56 for an example of dereferencing

Restrictions on Rowid Constraints:

- You cannot define a rowid constraint for the REF elements of a VARRAY column.
- You cannot subsequently drop a rowid constraint from a REF column.
- If the REF column or attribute is scoped, then this clause is ignored and the rowid is not stored with the REF value.

Referential Integrity Constraints on REF Columns

The *references_clause* of the *ref_constraint* syntax lets you define a foreign key constraint on the REF column.This clause also implicitly restricts the scope of the REF column or attribute to the referenced table. However, whereas a foreign key constraint on a non-REF column references an actual column in the parent table, a foreign key constraint on a REF column references the implicit object identifier (OID) column of the parent table.

If you do not specify CONSTRAINT, then Oracle generates a system name for the constraint of the form SYS_C*n*.

If you add a referential integrity constraint to an existing REF column that is already scoped, then the referenced table must be the same as the scope table of the REF column. If you later drop the referential integrity constraint, then the REF column will remain scoped to the referenced table.

As is the case for foreign key constraints on other types of columns, you can use the *references_clause* alone for inline declaration. For out-of-line declaration you must also specify the FOREIGN KEY keywords plus one or more REF columns or attributes.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for more information on object identifiers

Restrictions on Foreign Key Constraints of REF Columns:

- Oracle implicitly adds a scope constraint when you add a referential integrity constraint to an existing unscoped REF column. Therefore, all the restrictions that apply for scope constraints also apply in this case.
- You cannot specify a column after the object name in the *references_clause*.

Specifying Constraint State

As part of constraint definition, you can specify how and when Oracle should enforce the constraint.

constraint_state You can use the *constraint_state* with both inline and out-of-line specification. You can specify the clauses of *constraint_state* in any order, but you can specify each clause only once.

DEFERRABLE Clause The DEFERRABLE and NOT DEFERRABLE parameters indicate whether or not, in subsequent transactions, constraint checking can be

deferred until the end of the transaction using the SET CONSTRAINT(S) statement. If you omit this clause, then the default is NOT DEFERRABLE.

- Specify NOT DEFERRABLE to indicate that in subsequent transactions you cannot use the SET CONSTRAINT[S] clause to defer checking of this constraint until the transaction is committed. The checking of a NOT DEFERRABLE constraint can never be deferred to the end of the transaction.

> **Note:** If you declare a new constraint NOT DEFERRABLE, then it must be valid at the time the CREATE TABLE statement is committed or the CREATE statement will fail.

- Specify DEFERRABLE to indicate that in subsequent transactions you can use the SET CONSTRAINT[S] clause to defer checking of this constraint until after the transaction is committed. This setting in effect lets you disable the constraint temporarily while making changes to the database that might violate the constraint until all the changes are complete.

You cannot alter a constraint's deferrability. That is, whether you specify either of these parameters, or make the constraint NOT DEFERRABLE implicitly by specifying neither of them, you cannot specify this clause in an ALTER TABLE statement. You must drop the constraint and re-create it.

See Also:

- SET CONSTRAINT[S] on page 18-45 for information on setting constraint checking for a transaction
- *Oracle9i Database Administrator's Guide* and *Oracle9i Database Concepts* for more information about deferred constraints
- "DEFERRABLE Constraint Examples" on page 7-36

Restriction on the DEFERRABLE Clause: You cannot specify either of these parameters for a view constraint.

INITIALLY Clause The INITIALLY clause establishes the default checking behavior for constraints that are DEFERRABLE. The INITIALLY setting can be overridden by a SET CONSTRAINT(S) statement in a subsequent transaction.

- Specify INITIALLY IMMEDIATE to indicate that Oracle should check this constraint at the end of each subsequent SQL statement. If you do not specify INITIALLY at all, then the default is INITIALLY IMMEDIATE.

> **Note:** If you declare a new constraint INITIALLY IMMEDIATE, then it must be valid at the time the CREATE TABLE statement is committed or the create statement will fail.

- Specify INITIALLY DEFERRED to indicate that Oracle should check this constraint at the end of subsequent transactions.

This clause is not valid if you have declared the constraint to be NOT DEFERRABLE, because a NOT DEFERRABLE constraint is automatically INITIALLY IMMEDIATE and cannot ever be INITIALLY DEFERRED.

VALIDATE | NOVALIDATE The behavior of VALIDATE and NOVALIDATE always depends on whether the constraint is enabled or disabled, either explicitly or by default. Therefore they are described in the context of "ENABLE Clause" on page 7-20 and "DISABLE Clause" on page 7-21.

ENABLE Clause Specify ENABLE if you want the constraint to be applied to the data in the table.

> **Note:** If you enable a unique or primary key constraint, and if no index exists on the key, Oracle creates a unique index. This index is dropped if the constraint is subsequently disabled, and Oracle rebuilds the index every time the constraint is enabled.
>
> To avoid rebuilding the index and eliminate redundant indexes, create new primary key and unique constraints initially disabled. Then create (or use existing) nonunique indexes to enforce the constraint. Oracle does not drop a nonunique index when the constraint is disabled, so subsequent ENABLE operations are facilitated.

- ENABLE VALIDATE specifies that all old and new data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If any row in the table violates the integrity constraint, the constraint remains disabled and Oracle returns an error. If all rows comply with the constraint, Oracle enables the constraint. Subsequently, if new data violates the constraint, Oracle does not execute the statement and returns an error indicating the integrity constraint violation.

> **Note:** If you place a primary key constraint in ENABLE VALIDATE mode, the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before entering data into the column and before enabling the table's primary key constraint.

- ENABLE NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint. This clause does not ensure that existing data in the table complies with the constraint and therefore does not require a table lock.

If you specify neither VALIDATE nor NOVALIDATE, the default is VALIDATE.

If you change the state of any single constraint from ENABLE NOVALIDATE to ENABLE VALIDATE, the operation can be performed in parallel, and does not block reads, writes, or other DDL operations.

Restriction on the ENABLE Clause: You cannot enable a foreign key that references a disabled unique or primary key.

DISABLE Clause Specify DISABLE to disable the integrity constraint. Disabled integrity constraints appear in the data dictionary along with enabled constraints. If you do not specify this clause when creating a constraint, Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, because it lets you load large amounts of data while also saving space by not having an index. This setting lets you load data from a nonpartitioned table into a partitioned table using the *exchange_partition_clause* of the ALTER TABLE statement or using SQL*Loader. All other modifications to the table (inserts, updates, and deletes) by other SQL statements are disallowed.

  > **See Also:** *Oracle9i Data Warehousing Guide* for more information on using this setting

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated).

You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DISABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for information on when to use this setting

If you specify neither VALIDATE nor NOVALIDATE, the default is NOVALIDATE.

If you disable a unique or primary key constraint that is using a unique index, Oracle drops the unique index.

> **See Also:** the *enable_disable_clause* of CREATE TABLE on page 15-57 for additional notes and restrictions

RELY Clause RELY and NORELY are valid only when you are modifying an existing constraint (that is, in the ALTER TABLE ... MODIFY constraint syntax). These parameters specify whether a constraint in NOVALIDATE mode is to be taken into account for query rewrite. Specify RELY to activate an existing constraint in NOVALIDATE mode for query rewrite in an unenforced query rewrite integrity mode. The constraint is in NOVALIDATE mode, so Oracle does not enforce it. The default is NORELY.

Unenforced constraints are generally useful only with materialized views and query rewrite. Depending on the QUERY_REWRITE_INTEGRITY mode (see ALTER SESSION on page 10-2), query rewrite can use only constraints that are in VALIDATE mode, or that are in NOVALIDATE mode with the RELY parameter set, to determine join information.

Restriction on the RELY Clause: You cannot set a NOT NULL constraint to RELY.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information on materialized views and query rewrite

Using Indexes to Enforce Constraints

When defining the state of a unique or primary key constraint, you can specify an index for Oracle to use to enforce the constraint, or you can instruct Oracle to create the index used to enforce the constraint.

using_index_clause You can specify the *using_index_clause* only when enabling unique or primary key constraints. You can specify the clauses of the *using_index_clause* in any order, but you can specify each clause only once.

- If you specify *schema.index*, Oracle attempts to enforce the constraint using the specified index. If Oracle cannot find the index or cannot use the index to enforce the constraint, Oracle returns an error.
- If you specify the *create_index_statement*, Oracle attempts to create the index and use it to enforce the constraint. If Oracle cannot create the index or cannot use the index to enforce the constraint, Oracle returns an error.
- If you neither specify an existing index nor create a new index, Oracle creates the index. In this case:
  - The index receives the same name as the constraint.
  - You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, PCTFREE, and STORAGE parameters for the index. You cannot specify PCTUSED or the *logging_clause* for the index.
  - If *table* is partitioned, you can specify a locally or globally partitioned index for the unique or primary key constraint.

    See Also: "Explicit Index Control Example" on page 7-35 for an example of how you can create an index for Oracle to use in enforcing a constraint

Restrictions on the *using_index_clause*:

- You cannot specify this clause for a view constraint.
- You cannot specify this clause for a NOT NULL, foreign key, or check constraint.
- You cannot specify an index (*schema.index*) or create an index (*create_index_statement*) when enabling the primary key of an index-organized table.

  See Also:

  - *physical_attributes_clause* on page 7-52 for information on the INITRANS, MAXTRANS, TABLESPACE, and PCTFREE parameters and *storage_clause* on page 7-56 for information on the storage parameters
  - CREATE INDEX on page 13-62 for a description of LOCAL and *global_partitioned_index* clause, and for a description of NOSORT and *logging_clause* in relation to indexes
  - "Explicit Index Control Example" on page 7-35

NOSORT clause Specify NOSORT to indicate that the table rows are stored in the database in ascending order and that therefore Oracle does not have to sort the rows when creating the index.

logging_clause The *logging_clause* lets you specify whether creation of the index should be logged in the redo log file.

> **See Also:** *logging_clause* on page 7-45

global_partitioned_index The *global_partitioned_index* clause lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes. Oracle will partition the global index on the ranges of values from the table columns you specify in *column_list*. You cannot specify this clause for a local index.

The *column_list* must specify a left prefix of the index column list. That is, if the index is defined on columns a, b, and c, then for *column_list* you can specify (a, b, c), or (a, b), or (a, c), but you cannot specify (b, c) or (c) or (b, a).

Restrictions on *column_list*:

- You cannot specify more than 32 columns in *column_list*.
- The columns cannot contain the ROWID pseudocolumn or a column of type ROWID.

> **Note:** If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

> **See Also:** *Oracle9i Database Globalization Support Guide* for more information on character set support

index_partitioning_clause Use this clause to describe the individual index partitions. The number of repetitions of this clause determines the number of partitions. If you omit *partition*, Oracle generates a name with the form SYS_P*n*.

For VALUES LESS THAN (*value_list*), specify the (noninclusive) upper bound for the current partition in a global index. The value list is a comma-delimited, ordered list of literal values corresponding to the column list in the *global_*

partitioned_index clause. Always specify MAXVALUE as the value of the last partition.

> **Note:** If the index is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, you must use the TO_DATE function with a 4-character format mask for the year. The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT.

Handling Constraint Exceptions

When defining the state of a constraint, you can specify a table into which Oracle places the rowids of all rows violating the constraint.

exceptions_clause Use the *exceptions_clause* syntax to define exception handling. If you omit *schema*, then Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, then Oracle assumes that the table is named EXCEPTIONS. The EXCEPTIONS table or the table you specify must exist on your local database.

You can create the EXCEPTIONS table using one of these scripts:

- UTLEXCPT.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLEXPT1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own exceptions table, then it must follow the format prescribed by one of these two scripts.

Restrictions on the *exceptions_clause***:**

- You cannot specify this clause for a view constraint.
- You cannot specify this clause in a CREATE TABLE statement, because no rowids exist until *after* the successful completion of the statement.

Note: If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), then you must create a separate exceptions table for each index-organized table to accommodate its primary-key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

See Also:

- *Oracle9i Database Migration* for compatibility issues related to the use of these scripts
- The DBMS_IOT package in *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the SQL scripts
- *Oracle9i Database Performance Tuning Guide and Reference* for information on eliminating migrated and chained rows

View Constraints

Oracle does not enforce view constraints. However, operations on views are subject to the integrity constraints defined on the underlying base tables. This means that you can enforce constraints on views through constraints on base tables.

Restrictions on view constraints: View constraints are a subset of table constraints and are subject to the following restrictions:

- You can specify only unique, primary key, and foreign key constraints on views. However, you can define the view using the WITH CHECK OPTION clause, which is equivalent to specifying a check constraint for the view.
- Because view constraints are not enforced directly, you cannot specify INITIALLY DEFERRED or DEFERRABLE.
- View constraints are supported only in DISABLE NOVALIDATE mode. You must specify the keywords DISABLE NOVALIDATE when you declare the view constraint, and you cannot specify any other mode.
- You cannot specify the *using_index_clause*, the *exceptions_clause* clause, or the ON DELETE clause of the *references_clause*.
- You cannot define view constraints on attributes of an object column.

Examples

Unique Key Example The following statement is a variation of the statement that created the sample table `sh.promotions`. It defines inline and implicitly enables a unique key on the `promo_id` column (other constraints are not shown):

```
CREATE TABLE promotions_var1
    ( promo_id         NUMBER(6)
                       CONSTRAINT promo_id_u  UNIQUE
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    ) ;
```

The constraint `promo_id_u` identifies the `promo_id` column as a unique key. This constraint ensures that no two promotions in the table have the same ID. However, the constraint does allow promotions without identifiers.

Alternatively, you can define and enable this constraint out of line:

```
CREATE TABLE promotions_var2
    ( promo_id         NUMBER(6)
    , promo_name       VARCHAR2(20)
    , promo_category   VARCHAR2(15)
    , promo_cost       NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date   DATE
    , CONSTRAINT promo_id_u UNIQUE (promo_id)
   USING INDEX PCTFREE 20
      TABLESPACE stocks
      STORAGE (INITIAL 8K  NEXT 6K) );
```

The preceding statement also contains the *`using_index_clause`*, which specifies storage characteristics for the index that Oracle creates to enable the constraint.

Composite Unique Key Example The following statement defines and enables a composite unique key on the combination of the `warehouse_id` and `warehouse_name` columns of the `oe.warehouses` table:

```
ALTER TABLE warehouses
   ADD CONSTRAINT wh_unq UNIQUE (warehouse_id, warehouse_name)
   USING INDEX PCTFREE 5
   EXCEPTIONS INTO wrong_id;
```

The wh_unq constraint ensures that the same combination of warehouse_id and warehouse_name values does not appear in the table more than once.

The ADD CONSTRAINT clause also specifies other properties of the constraint:

- The USING INDEX clause specifies storage characteristics for the index Oracle creates to enable the constraint.
- The EXCEPTIONS INTO clause causes Oracle to write information to the wrong_id table about any rows currently in the customers table that violate the constraint. If the wrong_id exceptions table does not already exist, then this statement will fail.

Primary Key Example The following statement is a variation of the statement that created the sample table hr.locations. It creates the locations_demo table and defines and enables a primary key on the location_id column (other constraints from the hr.locations table are omitted):

```
CREATE TABLE locations_demo
    ( location_id    NUMBER(4) CONSTRAINT loc_id_pk PRIMARY KEY
    , street_address VARCHAR2(40)
    , postal_code    VARCHAR2(12)
    , city       VARCHAR2(30)
    , state_province VARCHAR2(25)
    , country_id     CHAR(2)
    ) ;
```

The loc_id_pk constraint, specified inline, identifies the location_id column as the primary key of the locations_demo table. This constraint ensures that no two locations in the table have the same location number and that no location identifier is NULL.

Alternatively, you can define and enable this constraint out of line:

```
CREATE TABLE locations_demo
    ( location_id    NUMBER(4)
    , street_address VARCHAR2(40)
    , postal_code    VARCHAR2(12)
    , city       VARCHAR2(30)
    , state_province VARCHAR2(25)
    , country_id     CHAR(2)
    , CONSTRAINT loc_id_pk PRIMARY KEY (location_id));
```

NOT NULL Example The following statement alters the `locations_demo` table (created in "Primary Key Example" on page 7-28) to define and enable a `NOT NULL` constraint on the `country_id` column:

```
ALTER TABLE locations_demo
   MODIFY (country_id CONSTRAINT country_nn NOT NULL);
```

The constraint `country_nn` ensures that no location in the table has a null `country_id`.

Composite Primary Key Example The following statement defines a composite primary key on the combination of the `prod_id` and `cust_id` columns of the sample table `sh.sales`:

```
ALTER TABLE sales
    ADD CONSTRAINT sales_pk PRIMARY KEY (prod_id, cust_id) DISABLE;
```

This constraint identifies the combination of the `prod_id` and `cust_id` columns as the primary key of the `sales` table. The constraint ensures that no two rows in the table have the same values for both the `prod_id` column and the `cust_id` column.

The constraint clause (`PRIMARY KEY`) also specifies the following properties of the constraint:

- The constraint definition does not include a constraint name, so Oracle generates a name for the constraint.
- The `DISABLE` clause causes Oracle to define the constraint but not enable it.

Foreign Key Constraint Example The following statement creates the `dept_20` table and defines and enables a foreign key on the `department_id` column that references the primary key on the `department_id` column of the `departments` table:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4),
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   CONSTRAINT fk_deptno
                    REFERENCES departments(department_id) );
```

The constraint fk_deptno ensures that all departments given for employees in the dept_20 table are present in the departments table. However, employees can have null department numbers, meaning they are not assigned to any department. To ensure that all employees are assigned to a department, you could create a NOT NULL constraint on the department_id column in the dept_20 table, in addition to the REFERENCES constraint.

Before you define and enable this constraint, you must define and enable a constraint that designates the department_id column of the departments table as a primary or unique key.

The foreign key constraint definition does not use the FOREIGN KEY clause, because the constraint is defined inline. The datatype of the *department_id* column is not needed, because Oracle automatically assigns to this column the datatype of the referenced key.

The constraint definition identifies both the parent table and the columns of the referenced key. Because the referenced key is the parent table's primary key, the referenced key column names are optional.

Alternatively, you can define this foreign key constraint out of line:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4),
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id,
   CONSTRAINT fk_deptno
      FOREIGN  KEY (department_id)
      REFERENCES  departments(department_id) );
```

The foreign key definitions in both statements of this statement omit the ON DELETE clause, causing Oracle to prevent the deletion of a department if any employee works in that department.

ON DELETE Example This statement creates the dept_20 table, defines and enables two referential integrity constraints, and uses the ON DELETE clause:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4) PRIMARY KEY,
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
```

```
    manager_id      NUMBER(4) CONSTRAINT fk_mgr
                    REFERENCES employees ON DELETE SET NULL,
    hire_date       DATE,
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   NUMBER(2)   CONSTRAINT fk_deptno
                    REFERENCES departments(department_id)
                    ON DELETE CASCADE );
```

Because of the first `ON DELETE` clause, if manager number 2332 is deleted from the `employees` table, then Oracle sets to null the value of `manager_id` for all employees in the `dept_20` table who previously had manager 2332.

Because of the second `ON DELETE` clause, Oracle cascades any deletion of a `department_id` value in the `departments` table to the `department_id` values of its dependent rows of the `dept_20` table. For example, if Department 20 is deleted from the `departments` table, then Oracle deletes all of that department's employees from the `dept_20` table.

Composite Foreign Key Constraint Example The following statement defines and enables a foreign key on the combination of the `employee_id` and `last_name` columns of the `dept_20` table:

```
ALTER TABLE dept_20
   ADD CONSTRAINT fk_empid_hiredate
   FOREIGN KEY (employee_id, hire_date)
   REFERENCES hr.job_history(employee_id, start_date)
   EXCEPTIONS INTO wrong_emp;
```

The constraint `fk_empid_empname` ensures that all the employees in the `dept_20` table have `employee_id` and `last_name` combinations that exist in the `employees` table. Before you define and enable this constraint, you must define and enable a constraint that designates the combination of the `employee_id` and `last_name` columns of the `employees` table as a primary or unique key.

The `EXCEPTIONS INTO` clause causes Oracle to write information to the `wrong_id` table about any rows in the `dept_20` table that violate the constraint. If the `wrong_id` exceptions table does not already exist, then this statement will fail.

Check Constraint Examples The following statement creates a `divisions` table and defines a `check` constraint in each of the table's columns:

```
CREATE TABLE divisions
   (div_no    NUMBER  CONSTRAINT check_divno
              CHECK (div_no BETWEEN 10 AND 99)
```

```
            DISABLE,
    div_name  VARCHAR2(9)  CONSTRAINT check_divname
            CHECK (div_name = UPPER(div_name))
            DISABLE,
    office    VARCHAR2(10)  CONSTRAINT check_office
            CHECK (office IN ('DALLAS','BOSTON',
            'PARIS','TOKYO'))
            DISABLE);
```

Each constraint restricts the values of the column in which it is defined:

- `check_divno` ensures that no division numbers are less than 10 or greater than 99.
- `check_divname` ensures that all department names are in uppercase.
- `check_office` restricts department locations to Dallas, Boston, Paris, or Tokyo.

Because each `CONSTRAINT` clause contains the `DISABLE` clause, Oracle only defines the constraints and does not enable them.

The following statement creates the `dept_20` table, defining out of line and implicitly enabling a check constraint:

```
CREATE TABLE dept_20
   (employee_id     NUMBER(4) PRIMARY KEY,
    last_name       VARCHAR2(10),
    job_id          VARCHAR2(9),
    manager_id      NUMBER(4),
    salary          NUMBER(7,2),
    commission_pct  NUMBER(7,2),
    department_id   NUMBER(2),
    CONSTRAINT check_sal CHECK (salary * commission_pct <= 5000));
```

This constraint uses an inequality condition to limit an employee's total commission, the product of `salary` and `commission_pct`, to $5000:

- If an employee has non-null values for both salary and commission, then the product of these values must not exceed $5000 to satisfy the constraint.
- If an employee has a null salary or commission, then the result of the condition is unknown and the employee automatically satisfies the constraint.

Because the constraint clause in this example does not supply a constraint name, Oracle generates a name for the constraint.

The following statement defines and enables a primary key constraint, two foreign key constraints, a NOT NULL constraint, and two check constraints:

```
CREATE TABLE order_detail
  (CONSTRAINT pk_od PRIMARY KEY (order_id, part_no),
   order_id    NUMBER
      CONSTRAINT fk_oid
         REFERENCES oe.orders(order_id),
   part_no     NUMBER
      CONSTRAINT fk_pno
         REFERENCES oe.product_information(product_id),
   quantity    NUMBER
      CONSTRAINT nn_qty NOT NULL
      CONSTRAINT check_qty CHECK (quantity > 0),
   cost        NUMBER
      CONSTRAINT check_cost CHECK (cost > 0) );
```

The constraints enable the following rules on table data:

- pk_od identifies the combination of the order_id and product_id columns as the primary key of the table. To satisfy this constraint, no two rows in the table can contain the same combination of values in the order_id and the product_id columns, and no row in the table can have a null in either the order_id or the product_id column.
- fk_oid identifies the order_id column as a foreign key that references the order_id column in the orders table in the sample schema oe. All new values added to the column order_detail.order_id must already appear in the column oe.orders.order_id.
- fk_pno identifies the product_id column as a foreign key that references the product_id column in the product_information table owned by oe. All new values added to the column order_detail.product_id must already appear in the column oe.product_information.product_id.
- nn_qty forbids nulls in the quantity column.
- check_qty ensures that values in the quantity column are always greater than zero.
- check_cost ensures the values in the cost column are always greater than zero.

This example also illustrates the following points about constraint clauses and column definitions:

- Out-of-line constraint definition can appear before or after the column definitions. In this example, the out-of-line definition of the pk_od constraint precedes the column definitions.
- A column definition can contain multiple inline constraint definitions. In this example, the definition of the quantity column contains the definitions of both the nn_qty and check_qty constraints.
- A table can have multiple CHECK constraints. Multiple CHECK constraints, each with a simple condition enforcing a single business rule, is better than a single CHECK constraint with a complicated condition enforcing multiple business rules. When a constraint is violated, Oracle returns an error identifying the constraint. Such an error more precisely identifies the violated business rule if the identified constraint enables a single business rule.

Attribute-Level Constraints Example The following example guarantees that a value exists for both the first_name and last_name attributes of the name column in the students table:

```
CREATE TYPE person_name AS OBJECT
   (first_name VARCHAR2(30), last_name VARCHAR2(30));
/

CREATE TABLE students (name person_name, age INTEGER,
   CHECK (name.first_name IS NOT NULL AND
          name.last_name IS NOT NULL));
```

REF Constraint Examples The following example creates a duplicate of the sample schema object type cust_address_typ, and then creates a table containing a REF column with a SCOPE constraint:

```
CREATE TYPE cust_address_typ_new AS OBJECT
    ( street_address     VARCHAR2(40)
    , postal_code        VARCHAR2(10)
    , city               VARCHAR2(30)
    , state_province     VARCHAR2(10)
    , country_id         CHAR(2)
    );
/
CREATE TABLE address_table OF cust_address_typ_new;

CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ_new
   SCOPE IS address_table);
```

The following example creates the same table but with a referential integrity constraint on the REF column that references the OID column of the parent table:

```
CREATE TABLE customer_addresses (
   add_id NUMBER,
   address REF cust_address_typ REFERENCES address_table);
```

The following example uses the type `department_typ` and the table `departments_obj_t` (created in "Creating Object Tables: Examples" on page 15-77). A table with a scoped REF is then created.

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ SCOPE IS departments_obj_t );
```

The following statement creates a table with a REF column which has a referential integrity constraint defined on it:

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ REFERENCES departments_obj_t);
```

Explicit Index Control Example The following statement shows another way to create a unique (or primary key) constraint that gives you explicit control over the index (or indexes) Oracle uses to enforce the constraint:

```
CREATE TABLE promotions_var3
    ( promo_id          NUMBER(6)
    , promo_name        VARCHAR2(20)
    , promo_category    VARCHAR2(15)
    , promo_cost        NUMBER(10,2)
    , promo_begin_date DATE
    , promo_end_date    DATE
    , CONSTRAINT promo_id_u UNIQUE (promo_id, promo_cost)
        USING INDEX (CREATE UNIQUE INDEX promo_ix1
           ON promotions_var3 (promo_id, promo_cost))
    , CONSTRAINT promo_id_u2 UNIQUE (promo_cost, promo_id)
        USING INDEX promo_ix1);
```

This example also shows that you can create an index for one constraint and use that index to create and enable another constraint in the same statement.

DEFERRABLE Constraint Examples The following statement creates table games with a NOT DEFERRABLE INITIALLY IMMEDIATE constraint check (by default) on the scores column:

```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

To define a unique constraint on a column as INITIALLY DEFERRED DEFERRABLE, issue the following statement:

```
CREATE TABLE games
  (scores NUMBER, CONSTRAINT unq_num UNIQUE (scores)
   INITIALLY DEFERRED DEFERRABLE);
```

deallocate_unused_clause

Purpose

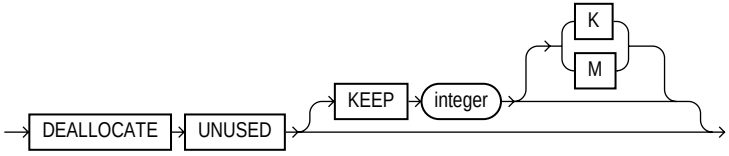
Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of a database object segment and make the space available for other segments in the tablespace.

You can deallocate unused space using the following statements:

- ALTER CLUSTER (see ALTER CLUSTER on page 9-7)
- ALTER INDEX: to deallocate unused space from the index, an index partition, or an index subpartition (see ALTER INDEX on page 9-64)
- ALTER MATERIALIZED VIEW: to deallocate unused space from the overflow segment of an index-organized materialized view (see ALTER MATERIALIZED VIEW on page 9-92)
- ALTER TABLE: to deallocate unused space from the table, a table partition, a table subpartition, the mapping table of an index-organized table, the overflow segment of an index-organized table, or a LOB storage segment (see ALTER TABLE on page 11-2)

Syntax

deallocate_unused_clause::=



Keywords and Parameters

This section describes the semantics of the *deallocate_unused_clause*. For additional information, refer to the SQL statement in which you set or reset this clause for a particular database object.

> **Note:** You cannot specify both the *deallocate_unused_clause* and the *allocate_extent_clause* in the same statement.

Oracle frees only unused space above the high water mark (that is, the point beyond which database blocks have not yet been formatted to receive data). Oracle deallocates unused space beginning from the end of the object and moving toward the beginning of the object to the high water mark.

If an extent is completely contained in the deallocation, then the whole extent is freed for reuse. If an extent is partially contained in the deallocation, then the used part up to the high water mark becomes the extent, and the remaining unused space is freed for reuse.

Oracle credits the amount of the released space to the user quota for the tablespace in which the deallocation occurs.

The exact amount of space freed depends on the values of the `INITIAL`, `MINEXTENTS`, and `NEXT` storage parameters.

> **See Also:** *storage_clause* on page 7-56 for a description of these parameters

KEEP *integer*

Specify the number of bytes above the high water mark that the segment of the database object is to have after deallocation.

- If you omit `KEEP` and the high water mark is above the size of `INITIAL` and `MINEXTENTS`, then all unused space above the high water mark is freed. When the high water mark is less than the size of `INITIAL` or `MINEXTENTS`, then all unused space above `MINEXTENTS` is freed.
- If you specify `KEEP`, then the specified amount of space is kept and the remaining space is freed. When the remaining number of extents is less than `MINEXTENTS`, then `MINEXTENTS` is adjusted to the new number of extents. If the initial extent becomes smaller than `INITIAL`, then `INITIAL` is adjusted to the new size.
- In either case, Oracle sets the value of the `NEXT` storage parameter to the size of the last extent that was deallocated.

file_specification

Purpose

Use one of the *file_specification* forms to specify a file as a datafile or tempfile, or to specify a group of one or more files as a redo log file group.

A *file_specification* can appear in the following statements:

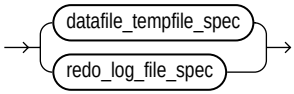
- CREATE CONTROLFILE (see CREATE CONTROLFILE on page 13-15)
- CREATE DATABASE (see CREATE DATABASE on page 13-22)
- ALTER DATABASE (see ALTER DATABASE on page 9-13)
- CREATE TABLESPACE (see CREATE TABLESPACE on page 15-80)
- CREATE TEMPORARY TABLESPACE (see CREATE TEMPORARY TABLESPACE on page 15-92)
- ALTER TABLESPACE (see ALTER TABLESPACE on page 11-101)

Prerequisites

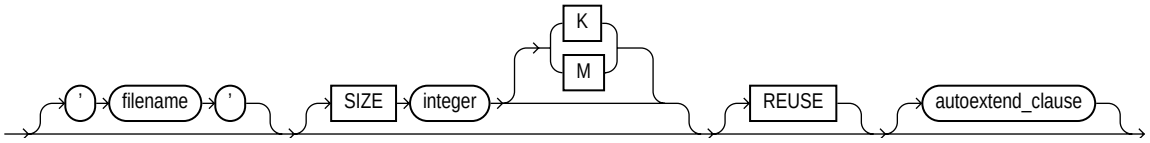
You must have the privileges necessary to issue one of the statements listed in the preceding section.

Syntax

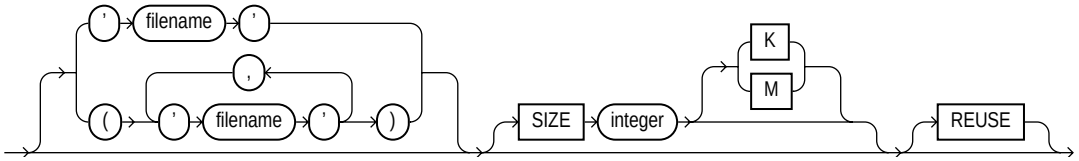
file_specification::=



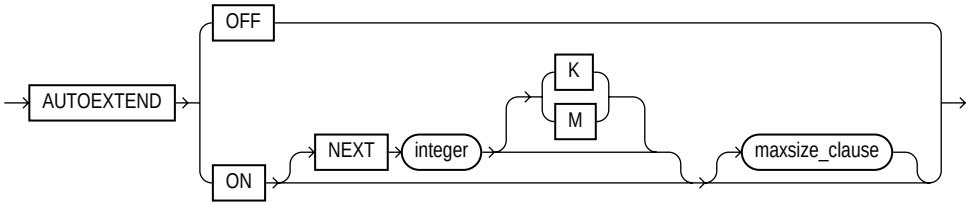
datafile_tempfile_spec::=



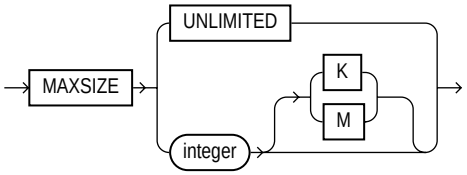
redo_log_file_spec::=



autoextend_clause::=



maxsize_clause::=



Keywords and Parameters

This section describes the semantics of *file_specification*. For additional information, refer to the SQL statement in which you specify a datafile, tempfile, or redo log file.

'filename'

For a *new* file, *filename* is the name of the new file. If you are not using Oracle-managed files, then you must specify *filename* or the statement fails. However, if you are using Oracle-managed files, then *filename* is optional, as are the remaining clauses of the specification. In this case, Oracle creates a unique name for the file and saves it in the directory specified by either the DB_CREATE_FILE_DEST initialization parameter (for any type of file) or by the DB_CREATE_ONLINE_LOG_DEST_*n* initialization parameter (which takes precedence over DB_CREATE_FILE_DEST for log files).

> **See Also:** *Oracle9i Database Administrator's Guide* for more information on Oracle-managed files, "Specifying a Datafile: Example" on page 7-43, and "Specifying a Log File: Example" on page 7-43

For an *existing* file, you must specify a filename. Specify the name of either a datafile, tempfile, or a redo log file member. The *filename* can contain only single-byte characters from 7-bit ASCII or EBCDIC character sets. Multibyte characters are not valid.

A redo log file group can have one or more members (copies). Each *filename* must be fully specified according to the conventions for your operating system.

SIZE Clause

Specify the size of the file in bytes. Use K or M to specify the size in kilobytes or megabytes.

- For undo tablespaces, you must specify the SIZE clause for each datafile. For other tablespaces, you can omit this parameter if the file already exists, or if you are creating an Oracle-managed file.
- If you omit this clause when creating an Oracle-managed file, then Oracle creates a 100M file.
- The size of a tablespace must be one block greater than the sum of the sizes of the objects contained in it.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on Automatic Undo Management and undo tablespaces and "Adding a Log File: Example" on page 7-43

REUSE

Specify REUSE to allow Oracle to reuse an existing file. You must specify REUSE if you specify a *filename* that already exists.

- If the file already exists, then Oracle reuses the filename and applies the new size (if you specify SIZE) or retains the original size.
- If the file does not exist, then Oracle ignores this clause and creates the file.

Restriction on the REUSE clause: You cannot specify REUSE unless you have specified *filename*.

Note: Whenever Oracle uses an existing file, the file's previous contents are lost.

See Also: "Adding a Datafile: Example" on page 7-44 and "Adding a Log File: Example" on page 7-43

autoextend_clause

Use the *autoextend_clause* to enable or disable the automatic extension of a new datafile or tempfile. If you omit this clause:

- For Oracle-managed files:
  - If you specify SIZE, Oracle creates a file of the specified size with AUTOEXTEND disabled.
  - If you do not specify SIZE, Oracle creates a 100M file with AUTOEXTEND enabled and MAXSIZE unlimited.
- For user-managed files, with or without SIZE specified, Oracle creates a file with AUTOEXTEND disabled.

ON Specify ON to enable autoextend.

OFF Specify OFF to turn off autoextend if is turned on.

Note: When you turn off autoextend, the values of NEXT and MAXSIZE are set to zero. If you turn autoextend back on in a subsequent statement, you must reset these values.

NEXT Use the NEXT clause to specify the size in bytes of the next increment of disk space to be allocated automatically when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is the size of one data block.

MAXSIZE Use the MAXSIZE clause to specify the maximum disk space allowed for automatic extension of the datafile.

UNLIMITED Use the UNLIMITED clause if you do not want to limit the disk space that Oracle can allocate to the datafile or tempfile.

Restriction on the *autoextend_clause*: You cannot specify this clause as part of *datafile_tempfile_spec* in a CREATE CONTROLFILE statement or in an ALTER DATABASE CREATE DATAFILE clause.

Examples

Specifying a Log File: Example The following statement creates a database named payable that has two redo log file groups, each with two members, and one datafile:

```
CREATE DATABASE payable
   LOGFILE GROUP 1 ('diska:log1.log', 'diskb:log1.log') SIZE 50K,
           GROUP 2 ('diska:log2.log', 'diskb:log2.log') SIZE 50K
   DATAFILE 'diskc:dbone.dat' SIZE 30M;
```

The first file specification in the LOGFILE clause specifies a redo log file group with the GROUP value 1. This group has members named 'diska:log1.log' and 'diskb:log1.log', each 50 kilobytes in size.

The second file specification in the LOGFILE clause specifies a redo log file group with the GROUP value 2. This group has members named 'diska:log2.log' and 'diskb:log2.log', also 50 kilobytes in size.

The file specification in the DATAFILE clause specifies a datafile named 'diskc:dbone.dat', 30 megabytes in size.

Each file specification specifies a value for the SIZE parameter and omits the REUSE clause, so none of these files can already exist. Oracle must create them.

Adding a Log File: Example The following statement adds another redo log file group with two members to the payable database:

```
ALTER DATABASE payable
   ADD LOGFILE GROUP 3 ('diska:log3.log', 'diskb:log3.log')
   SIZE 50K REUSE;
```

The file specification in the ADD LOGFILE clause specifies a new redo log file group with the GROUP value 3. This new group has members named 'diska:log3.log' and 'diskb:log3.log', each 50 kilobytes in size. Because the file specification specifies the REUSE clause, each member can (but need not) already exist.

Specifying a Datafile: Example The following statement creates a tablespace named stocks that has three datafiles:

```
CREATE TABLESPACE stocks
```

```
   DATAFILE 'stock1.dat' SIZE 10M,
            'stock2.dat' SIZE 10M,
            'stock3.dat' SIZE 10M;
```

The file specifications for the datafiles specify files named 'diskc:stock1.dat', 'diskc:stock2.dat', and 'diskc:stock3.dat'.

Adding a Datafile: Example The following statement alters the stocks tablespace and adds a new datafile:

```
ALTER TABLESPACE stocks
   ADD DATAFILE 'stock4.dat' SIZE 10M REUSE;
```

The file specification specifies a datafile named 'diskc:stock4.dat'. If the filename does not exist, then Oracle simply ignores the REUSE keyword.

logging_clause

Purpose

The *logging_clause* lets you specify whether creation of a database object will be logged in the redo log file (LOGGING) or not (NOLOGGING).

This clause also specifies whether subsequent Direct Loader (SQL*Loader) and direct-path INSERT operations against the object are logged (LOGGING) or not logged (NOLOGGING).

You can specify the *logging_clause* in the following statements:

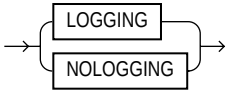
- CREATE TABLE and ALTER TABLE: for logging of the table, one of its partitions, a LOB segment, or the overflow segment of an index-organized table (see CREATE TABLE on page 15-7 and ALTER TABLE on page 11-2).
- CREATE INDEX and ALTER INDEX: for logging of the index or one of its partitions (see CREATE INDEX on page 13-62 and ALTER INDEX on page 9-64).
- CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW: for logging of the materialized view, one of its partitions, or a LOB segment (see CREATE MATERIALIZED VIEW on page 14-5 and ALTER MATERIALIZED VIEW on page 9-92).
- CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG: for logging of the materialized view log or one of its partitions (see CREATE MATERIALIZED VIEW LOG on page 14-32 and ALTER MATERIALIZED VIEW LOG on page 9-112.)
- CREATE TABLESPACE and ALTER TABLESPACE: to set or modify the default logging characteristics for all objects created in the tablespace (see CREATE TABLESPACE on page 15-80 and ALTER TABLESPACE on page 11-101.)

You can also specify LOGGING or NOLOGGING for the following operations:

- Rebuilding an index (using CREATE INDEX ... REBUILD)
- Moving a table (using ALTER TABLE ... MOVE)

Syntax

logging_clause::=



Keywords and Parameters

This section describes the semantics of the *logging_clause*. For additional information, refer to the SQL statement in which you set or reset logging characteristics for a particular database object.

Specify LOGGING if you want the creation of a database object, as well as subsequent inserts into the object, to be logged in the redo log file.

Specify NOLOGGING if you do not want these operations to be logged.

- For a **nonpartitioned object**, the value specified for this clause is the actual physical attribute of the segment associated with the object.
- For **partitioned objects**, the value specified for this clause is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and in subsequent ALTER ... ADD PARTITION statements), unless you specify the logging attribute in the PARTITION description.

If the object for which you are specifying the logging attributes resides in a database or tablespace in force logging mode, then Oracle ignores any NOLOGGING setting until the database or tablespace is taken out of force logging mode.

If the database is run in ARCHIVELOG mode, then media recovery from a backup made before the LOGGING operation re-creates the object. However, media recovery from a backup made before the NOLOGGING operation does not re-create the object.

The size of a redo log generated for an operation in NOLOGGING mode is significantly smaller than the log generated in LOGGING mode.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not fully logged. Therefore, if you cannot afford to lose the database object, then you should take a backup after the NOLOGGING operation.

NOLOGGING is supported in only a subset of the clauses that support LOGGING. Only the following operations support the NOLOGGING mode:

DML:

- Direct-path INSERT (serial or parallel)
- Direct Loader (SQL*Loader)

DDL:

- CREATE TABLE ... AS SELECT
- CREATE TABLE ... *LOB_storage_clause* ... *LOB_parameters* ... NOCACHE | CACHE READS
- ALTER TABLE ... *LOB_storage_clause* ... *LOB_parameters* ... NOCACHE | CACHE READS (for specify logging of newly created LOB columns)
- ALTER TABLE ... *modify_LOB_storage_clause* ... *modify_LOB_parameters* ... NOCACHE | CACHE READS (to change logging of existing LOB columns)
- ALTER TABLE ... MOVE
- ALTER TABLE ... [all partition operations that involve data movement]
  - ALTER TABLE ... ADD PARTITION (hash partition only)
  - ALTER TABLE ... MERGE PARTITIONS
  - ALTER TABLE ... SPLIT PARTITION
  - ALTER TABLE ... MOVE PARTITION
  - ALTER TABLE ... MODIFY PARTITION ... ADD SUBPARTITION
  - ALTER TABLE ... MODIFY PARTITION ... COALESCE SUBPARTITION
  - ALTER TABLE ... MODIFY PARTITION ... REBUILD UNUSABLE INDEXES
- CREATE INDEX
- ALTER INDEX ... REBUILD
- ALTER INDEX ... REBUILD PARTITION
- ALTER INDEX ... SPLIT PARTITION

For **objects other than LOBs**, if you omit this clause, then the logging attribute of the object defaults to the logging attribute of the tablespace in which it resides.

For **LOBs**, if you omit this clause:

- If you specify CACHE, then LOGGING is used (because you cannot have CACHE NOLOGGING).

- If you specify NOCACHE or CACHE READS, then the logging attribute defaults to the logging attribute of the tablespace in which it resides.

NOLOGGING does not apply to LOBs that are stored inline with row data. That is, if you specify NOLOGGING for LOBs with values less than 4000 bytes and you have not disabled STORAGE IN ROW, then Oracle ignores the NOLOGGING specification and treats the LOB data the same as other table data.

> **See Also:** *Oracle9i Database Concepts* and *Oracle9i Database Administrator's Guide* for more information about logging and parallel DML

parallel_clause

Purpose

The *parallel_clause* lets you parallelize the creation of a database object and set the default degree of parallelism for subsequent queries of and DML operations on the object.

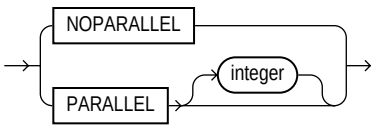
You can specify the *parallel_clause* in the following statements:

- CREATE TABLE: to set parallelism for the table (see CREATE TABLE on page 15-7).
- ALTER TABLE (see ALTER TABLE on page 11-2):
  - To change parallelism for the table
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a table partition
- CREATE CLUSTER and ALTER CLUSTER: to set or alter parallelism for a cluster (see CREATE CLUSTER on page 13-2 and ALTER CLUSTER on page 9-7).
- CREATE INDEX: to set parallelism for the index (see CREATE INDEX on page 13-62).
- ALTER INDEX (see ALTER INDEX on page 9-64):
  - To change parallelism for the table
  - To parallelize the rebuilding of the index or the splitting of an index partition
- CREATE MATERIALIZED VIEW: to set parallelism for the materialized view (see CREATE MATERIALIZED VIEW on page 14-5).
- ALTER MATERIALIZED VIEW (see ALTER MATERIALIZED VIEW on page 9-92):
  - To change parallelism for the materialized view
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a materialized view partition
  - To parallelize the operations of adding or moving materialized view subpartitions

- CREATE MATERIALIZED VIEW LOG: to set parallelism for the table (see CREATE MATERIALIZED VIEW LOG on page 14-32).
- ALTER MATERIALIZED VIEW LOG (see ALTER MATERIALIZED VIEW LOG on page 9-112):
  - To change parallelism for the materialized view
  - To parallelize the operations of adding, coalescing, exchanging, merging, splitting, truncating, dropping, or moving a materialized view partition
- ALTER DATABASE ... RECOVER: to recover the database (see ALTER DATABASE on page 9-13).
- ALTER DATABASE ... *standby_database_clauses*: to parallelize operations on the standby database (see ALTER DATABASE on page 9-13).

Syntax

parallel_clause::=



Keywords and Parameters

This section describes the semantics of the *parallel_clause*. For additional information, refer to the SQL statement in which you set or reset parallelism for a particular database object or operation.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

physical_attributes_clause

Purpose

The *physical_attributes_clause* lets you specify the value of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics of a table, cluster, index, or materialized view.

You can specify the *physical_attributes_clause* in the following statements:

- CREATE CLUSTER and ALTER CLUSTER: to set or change the physical attributes of the cluster and all tables in the cluster (see CREATE CLUSTER on page 13-2 and ALTER CLUSTER on page 9-7).
- CREATE TABLE: to set the physical attributes of the table, a table partition, the OIDINDEX of an object table, or the overflow segment of an index-organized table (see CREATE TABLE on page 15-7).
- ALTER TABLE: to change the physical attributes of the table, the default physical attributes of future table partitions, or the physical attributes of existing table partitions (see ALTER TABLE on page 11-2).

  Notes:

  - You cannot specify physical attributes for a temporary table.
  - You cannot specify physical attributes for a clustered table. Tables in a cluster inherit the cluster's physical attributes.

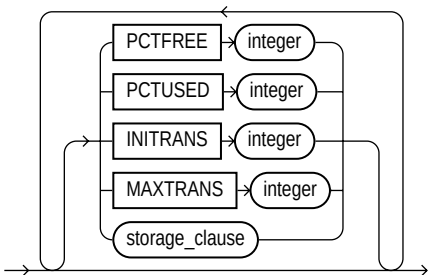
- CREATE INDEX: to set the physical attributes of an index, or index partition (see CREATE INDEX on page 13-62).
- ALTER INDEX: to change the physical attributes of the index, the default physical attributes of future partitions, or the physical attributes of existing index partitions (see ALTER INDEX on page 9-64).
- CREATE MATERIALIZED VIEW: to set the physical attributes of the materialized view, one of its partitions, or the index Oracle generates to maintain the materialized view (see CREATE MATERIALIZED VIEW on page 14-5).
- ALTER MATERIALIZED VIEW: to change the physical attributes of the materialized view, the default physical attributes of future partitions, the physical attributes of an existing partition, or the index Oracle creates to

maintain the materialized view (see ALTER MATERIALIZED VIEW on page 9-92).

- CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG: to set or change the physical attributes of the materialized view log (see CREATE MATERIALIZED VIEW LOG on page 14-32 and ALTER MATERIALIZED VIEW LOG on page 9-112).

Syntax

physical_attributes_clause::=



(*storage_clause* on page 7-56)

Keywords and Parameters

This section describes the parameters of the *physical_attributes_clause*. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

PCTFREE integer

Specify a whole number representing the percentage of space in each data block of the database object reserved for future updates to the object's rows. The value of PCTFREE must be a value from 0 to 99. A value of 0 means that the entire block can be filled by inserts of new rows. The default value is 10. This value reserves 10% of each block for updates to existing rows and allows inserts of new rows to fill a maximum of 90% of each block.

PCTFREE has the same function in the statements that create and alter tables, partitions, clusters, indexes, materialized views, and materialized view logs. The combination of PCTFREE and PCTUSED determines whether new rows will be inserted into existing data blocks or into new blocks.

Restriction on the PCTFREE clause: When altering an index, you can specify this parameter only in the *modify_index_default_attrs* clause and the *split_partition_clause*.

PCTUSED integer

Specify a whole number representing the minimum percentage of used space that Oracle maintains for each data block of the database object. A block becomes a candidate for row insertion when its used space falls below PCTUSED. PCTUSED is specified as a positive integer from 0 to 99 and defaults to 40.

PCTUSED has the same function in the statements that create and alter tables, partitions, clusters, materialized views, and materialized view logs.

PCTUSED is not a valid table storage characteristic for an index-organized table (ORGANIZATION INDEX).

The sum of PCTFREE and PCTUSED must be equal to or less than 100. You can use PCTFREE and PCTUSED together to utilize space within a table more efficiently.

Restrictions on the PCTUSED clause: You cannot specify this parameter for an index or for the index segment of an index-organized table.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for information on the performance effects of different values PCTUSED and PCTFREE

INITRANS integer

Specify the initial number of concurrent transaction entries allocated within each data block allocated to the database object. This value can range from 1 to 255 and defaults to 1, with the following exceptions:

- The default INITRANS value for a cluster or index is 2 or the default INITRANS value of the cluster's tablespace, whichever is greater.
- The default value for an index is 2

In general, you should not change the INITRANS value from its default.

Each transaction that updates a block requires a transaction entry in the block. The size of a transaction entry depends on your operating system.

This parameter ensures that a minimum number of concurrent transactions can update the block and helps avoid the overhead of dynamically allocating a transaction entry.

The INITRANS parameter serves the same purpose in the statements that create and alter tables, partitions, clusters, indexes, materialized views, and materialized view logs.

MAXTRANS integer

Specify the maximum number of concurrent transactions that can update a data block allocated to the database object. This limit does not apply to queries. This value can range from 1 to 255 and the default is a function of the data block size. You should not change the MAXTRANS value from its default.

If the number of concurrent transactions updating a block exceeds the INITRANS value, then Oracle dynamically allocates transaction entries in the block until either the MAXTRANS value is exceeded or the block has no more free space.

The MAXTRANS parameter serves the same purpose in the PARTITION description, clusters, materialized views, and materialized view logs as in tables.

storage_clause The *storage_clause* lets you specify storage characteristics for the table, object table OID index, partition, LOB data segment, LOB index segment, or index-organized table overflow data segment. This clause has performance ramifications for large tables. Storage should be allocated to minimize dynamic allocation of additional space.

See Also: *storage_clause* on page 7-56

storage_clause

Purpose

The *storage_clause* lets you specify how Oracle should store a database object. Storage parameters affect both how long it takes to access data stored in the database and how efficiently space in the database is used.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for a discussion of the effects of the storage parameters

When you create a cluster, index, rollback segment, materialized view, materialized view log, table, or partition, you can specify values for the storage parameters for the segments allocated to these objects. If you omit any storage parameter, then Oracle uses the value of that parameter specified for the tablespace in which the object resides.

When you alter a cluster, index, rollback segment, materialized view, materialized view log, table, or partition, you can change the values of storage parameters. The new values affect only future extent allocations.

The *storage_clause* is part of the *physical_attributes_clause*, so you can specify this clause in any of the statements where you can specify the physical attributes clause (see *physical_attributes_clause* on page 7-52).

In addition, you can specify the *storage_clause* in the following statements:

- CREATE CLUSTER and ALTER CLUSTER: to set or change the storage characteristics of the cluster and all tables in the cluster (see CREATE CLUSTER on page 13-2 and ALTER CLUSTER on page 9-7).
- CREATE INDEX and ALTER INDEX: to set or change the storage characteristics of an index or index partition (see CREATE INDEX on page 13-62 and ALTER INDEX on page 9-64).
- CREATE MATERIALIZED VIEW and ALTER MATERIALIZED VIEW: to set or change the storage characteristics of a materialized view, one of its partitions, or the index Oracle generates to maintain the materialized view (see CREATE MATERIALIZED VIEW on page 14-5 and ALTER MATERIALIZED VIEW on page 9-92).
- CREATE MATERIALIZED VIEW LOG and ALTER MATERIALIZED VIEW LOG: to set or change the storage characteristics of the materialized view log (see

CREATE MATERIALIZED VIEW LOG on page 14-32 and ALTER MATERIALIZED VIEW LOG on page 9-112).

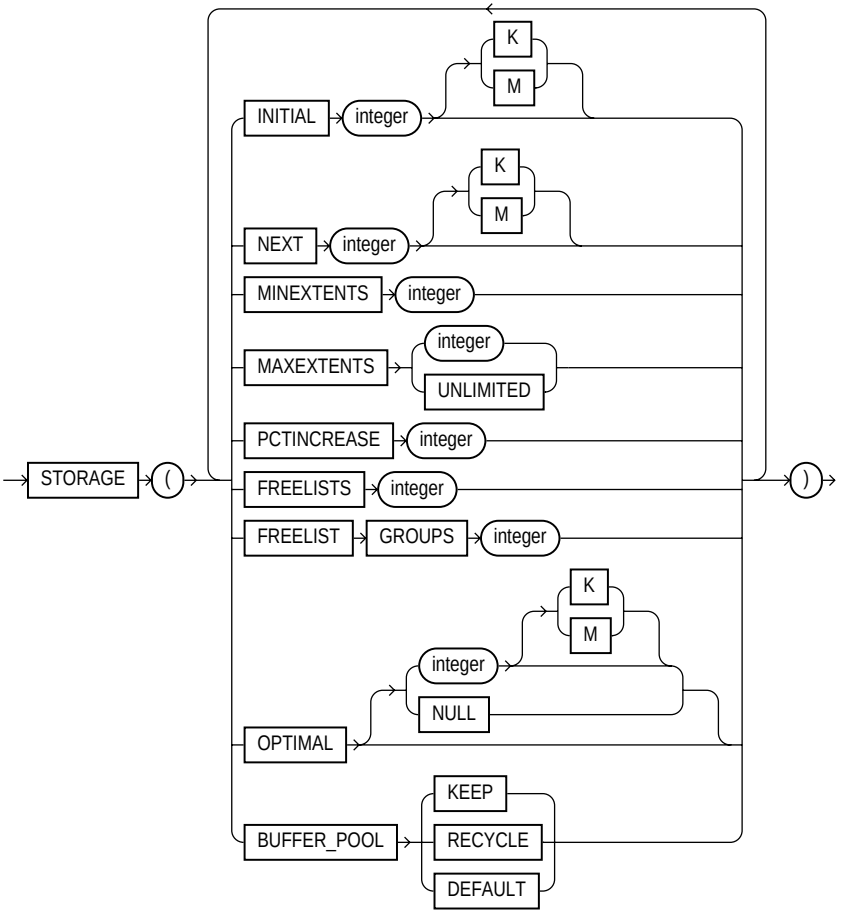
- CREATE ROLLBACK SEGMENT and ALTER ROLLBACK SEGMENT: to set or change the storage attributes of a rollback segment (see CREATE ROLLBACK SEGMENT on page 14-80 and ALTER ROLLBACK SEGMENT on page 9-138).
- CREATE TABLE and ALTER TABLE: to set the storage characteristics of a LOB data segment of the table or one of its partitions or subpartitions (see CREATE TABLE on page 15-7 and ALTER TABLE on page 11-2).
- CREATE TABLESPACE and ALTER TABLESPACE: to set or change the default storage characteristics for objects created in the tablespace (see CREATE TABLESPACE on page 15-80 and ALTER TABLESPACE on page 11-101).
- *constraints*: to specify storage for the index (and its partitions, if it is a partitioned index) used to enforce the constraint (see *constraints* on page 7-5).

Prerequisites

To change the value of a STORAGE parameter, you must have the privileges necessary to use the appropriate CREATE or ALTER statement.

Syntax

storage_clause::=



Keywords and Parameters

This section describes the parameters of the *storage_clause*. For additional information, refer to the SQL statement in which you set or reset these parameters for a particular database object.

Note: The *storage_clause* is interpreted differently for locally managed tablespaces. At creation, Oracle ignores MAXEXTENTS and uses the remaining parameter values to calculate the initial size of the segment. For more information, see CREATE TABLESPACE on page 15-80.

See Also: "Specifying Table Storage Attributes: Example" on page 7-64 and "Specifying Rollback Segment Storage Attributes: Example" on page 7-65

INITIAL

Specify in bytes the size of the object's first extent. Oracle allocates space for this extent when you create the schema object. Use K or M to specify this size in kilobytes or megabytes.

The default value is the size of 5 data blocks. In tablespaces with manual segment-space management, the minimum value is the size of 2 data blocks plus one data block for each free list group you specify. In tablespaces with automatic segment-space management, the minimum value is 5 data blocks. The maximum value depends on your operating system.

In dictionary-managed tablespaces, if MINIMUM EXTENT was specified for the tablespace when it was created, then Oracle rounds the value of INITIAL up to the specified MINIMUM EXTENT size if necessary. If MINIMUM EXTENT was not specified, then Oracle rounds the INITIAL extent size for segments created in that tablespace up to the minimum value (see preceding paragraph), or to multiples of 5 blocks if the requested size is greater than 5 blocks.

In locally managed tablespaces, Oracle uses the value of INITIAL in conjunction with the size of extents specified for the tablespace to determine the object's first extent. For example, in a uniform locally managed tablespace with 5M extents, if you specify an INITIAL value of 1M, then Oracle creates five 1M extents.

Restriction on INITIAL: You cannot specify INITIAL in an ALTER statement.

See Also: FREELIST GROUPS on page 7-61 for information on freelist groups

NEXT

Specify in bytes the size of the next extent to be allocated to the object. Use K or M to specify the size in kilobytes or megabytes. The default value is the size of 5 data

blocks. The minimum value is the size of 1 data block. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation, as described in *Oracle9i Database Administrator's Guide*.

If you change the value of the NEXT parameter (that is, if you specify it in an ALTER statement), then the next allocated extent will have the specified size, regardless of the size of the most recently allocated extent and the value of the PCTINCREASE parameter.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on how Oracle minimizes fragmentation

PCTINCREASE

Specify the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, meaning that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system.

Oracle rounds the calculated size of each new extent to the nearest multiple of the data block size.

If you change the value of the PCTINCREASE parameter (that is, if you specify it in an ALTER statement), then Oracle calculates the size of the next extent using this new value and the size of the most recently allocated extent.

> **Suggestion:** If you wish to keep all extents the same size, you can prevent SMON from coalescing extents by setting the value of PCTINCREASE to 0. In general, Oracle Corporation recommends a setting of 0 as a way to minimize fragmentation and avoid the possibility of very large temporary segments during processing.

Restriction on PCTINCREASE: You cannot specify PCTINCREASE for rollback segments. Rollback segments always have a PCTINCREASE value of 0.

MINEXTENTS

Specify the total number of extents to allocate when the object is created. This parameter lets you allocate a large amount of space when you create an object, even if the space available is not contiguous. The default and minimum value is 1,

meaning that Oracle allocates only the initial extent, except for rollback segments, for which the default and minimum value is 2. The maximum value depends on your operating system.

If the MINEXTENTS value is greater than 1, then Oracle calculates the size of subsequent extents based on the values of the INITIAL, NEXT, and PCTINCREASE storage parameters.

When changing the value of MINEXTENTS (that is, in an ALTER statement), you can reduce the value from its current value, but you cannot increase it. Resetting MINEXTENTS to a smaller value might be useful, for example, before a TRUNCATE ... DROP STORAGE statement, if you want to ensure that the segment will maintain a minimum number of extents after the TRUNCATE operation.

Restriction on MINEXTENTS: You cannot change the value of MINEXTENTS for an object that resides in a locally managed tablespace.

MAXEXTENTS

Specify the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1 (except for rollback segments, which always have a minimum value of 2). The default value depends on your data block size.

Restriction on MAXEXTENTS: You cannot change the value of MAXEXTENTS for an object that resides in a locally managed tablespace.

UNLIMITED Specify UNLIMITED if you want extents to be allocated automatically as needed. Oracle Corporation recommends this setting as a way to minimize fragmentation.

However, do not use this clause for rollback segments. Rogue transactions containing inserts, updates, or deletes that continue for a long time will continue to create new extents until a disk is full.

Caution: A rollback segment that you create without specifying the *storage_clause* has the same storage parameters as the tablespace in which the rollback segment is created. Thus, if you create the tablespace with MAXEXTENTS UNLIMITED, then the rollback segment will also have the same default.

FREELIST GROUPS

Specify the number of groups of free lists for the database object you are creating. The default and minimum value for this parameter is 1. Oracle uses the instance

number of Real Application Clusters instances to map each instance to one free list group.

Each free list group uses one database block. Therefore:

- If you do not specify a large enough value for INITIAL to cover the minimum value plus one data block for each free list group, then Oracle increases the value of INITIAL the necessary amount.
- If you are creating an object in a uniform locally managed tablespace, and the extent size is not large enough to accommodate the number of freelist groups, then the create operation will fail.

> **Note:** Oracle ignores a setting of FREELIST GROUPS if the tablespace in which the object resides is in automatic segment-space management mode. If you are running your database in this mode, please refer to the FREEPOOLS parameter of the *LOB_storage_clause* on page 15-37.

Restriction on FREELIST GROUPS: You can specify the FREELIST GROUPS parameter only in CREATE TABLE, CREATE CLUSTER, CREATE MATERIALIZED VIEW, CREATE MATERIALIZED VIEW LOG, and CREATE INDEX statements.

> **See Also:** *Oracle9i Real Application Clusters Administration*

FREELISTS

For objects other than tablespaces, specify the number of free lists for each of the free list groups for the table, partition, cluster, or index. The default and minimum value for this parameter is 1, meaning that each free list group contains one free list. The maximum value of this parameter depends on the data block size. If you specify a FREELISTS value that is too large, then Oracle returns an error indicating the maximum value.

> **Note:** Oracle ignores a setting of FREELISTS if the tablespace in which the object resides is in automatic segment-space management mode. If you are running your database in this mode, please refer to the FREEPOOLS parameter of the *LOB_storage_clause* on page 15-37.

Restriction on FREELISTS: You can specify FREELISTS in the *storage_clause* of any statement except when creating or altering a tablespace or rollback segment.

OPTIMAL

The OPTIMAL keyword is relevant only to rollback segments. It specifies an optimal size in bytes for a rollback segment. Use K or M to specify this size in kilobytes or megabytes. Oracle tries to maintain this size for the rollback segment by dynamically deallocating extents when their data is no longer needed for active transactions. Oracle deallocates as many extents as possible without reducing the total size of the rollback segment below the OPTIMAL value.

The value of OPTIMAL cannot be less than the space initially allocated by the MINEXTENTS, INITIAL, NEXT, and PCTINCREASE parameters. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size.

NULL Specify NULL for no optimal size for the rollback segment, meaning that Oracle never deallocates the rollback segment's extents. This is the default behavior.

BUFFER_POOL

The BUFFER_POOL clause lets you specify a default buffer pool (cache) for a schema object. All blocks for the object are stored in the specified cache.

- If you define a buffer pool for a partitioned table or index, then the partitions inherit the buffer pool from the table or index definition, unless overridden by a partition-level definition.
- For an index-organized table, you can specify a buffer pool separately for the index segment and the overflow segment.

Restrictions on BUFFER_POOL:

- You cannot specify this clause for a cluster table. However, you can specify it for a cluster.
- You cannot specify this clause for a tablespace or for a rollback segment.

KEEP Specify KEEP to put blocks from the segment into the KEEP buffer pool. Maintaining an appropriately sized KEEP buffer pool lets Oracle retain the schema object in memory to avoid I/O operations. KEEP takes precedence over any NOCACHE clause you specify for a table, cluster, materialized view, or materialized view log.

RECYCLE Specify RECYCLE to put blocks from the segment into the RECYCLE pool. An appropriately sized RECYCLE pool reduces the number of objects whose default pool is the RECYCLE pool from taking up unnecessary cache space.

DEFAULT Specify DEFAULT to indicate the default buffer pool. This is the default for objects not assigned to KEEP or RECYCLE.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for more information about using multiple buffer pools

Examples

Specifying Table Storage Attributes: Example The following statement creates a table and provides storage parameter values:

```
CREATE TABLE divisions
    (div_no     NUMBER(2),
     div_name   VARCHAR2(14),
     location   VARCHAR2(13) )
     STORAGE  ( INITIAL 100K  NEXT     50K
                MINEXTENTS 1  MAXEXTENTS 50  PCTINCREASE 5);
```

Oracle allocates space for the table based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 1, so Oracle allocates 1 extent for the table upon creation.
- The INITIAL value is 100K, so the first extent's size is 100 kilobytes.
- If the table data grows to exceed the first extent, then Oracle allocates a second extent. The NEXT value is 50K, so the second extent's size would be 50 kilobytes.
- If the table data subsequently grows to exceed the first two extents, then Oracle allocates a third extent. The PCTINCREASE value is 5, so the calculated size of the third extent is 5% larger than the second extent, or 52.5 kilobytes. If the data block size is 2 kilobytes, then Oracle rounds this value to 52 kilobytes.

  If the table data continues to grow, then Oracle allocates more extents, each 5% larger than the previous one.
- The MAXEXTENTS value is 50, so Oracle can allocate as many as 50 extents for the table.

Specifying Rollback Segment Storage Attributes: Example The following statement creates a rollback segment and provides storage parameter values:

```
CREATE ROLLBACK SEGMENT rs_store
    STORAGE ( INITIAL  10K  NEXT 10K
              MINEXTENTS 2  MAXEXTENTS 25
               OPTIMAL 50K );
```

Oracle allocates space for the rollback segment based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 2, so Oracle allocates 2 extents for the rollback segment upon creation.
- The INITIAL value is 10K, so the first extent's size is 10 kilobytes.
- The NEXT value is 10K, so the second extent's size is 10 kilobytes.
- If the rollback data exceeds the first two extents, then Oracle allocates a third extent. The PCTINCREASE value for rollback segments is always 0, so the third and subsequent extents are the same size as the second extent, 10 kilobytes.
- The MAXEXTENTS value is 25, so Oracle can allocate as many as 25 extents for the rollback segment.
- The OPTIMAL value is 50K, so Oracle deallocates extents if the rollback segment exceeds 50 kilobytes. Oracle deallocates only extents that contain data for transactions that are no longer active.

8

SQL Queries and Subqueries

This chapter describes SQL queries and subqueries. This chapter contains these sections:

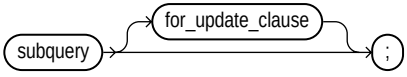
About Queries and Subqueries

A **query** is an operation that retrieves data from one or more tables or views. In this reference, a top-level SELECT statement is called a **query**, and a query nested within another SQL statement is called a **subquery**.

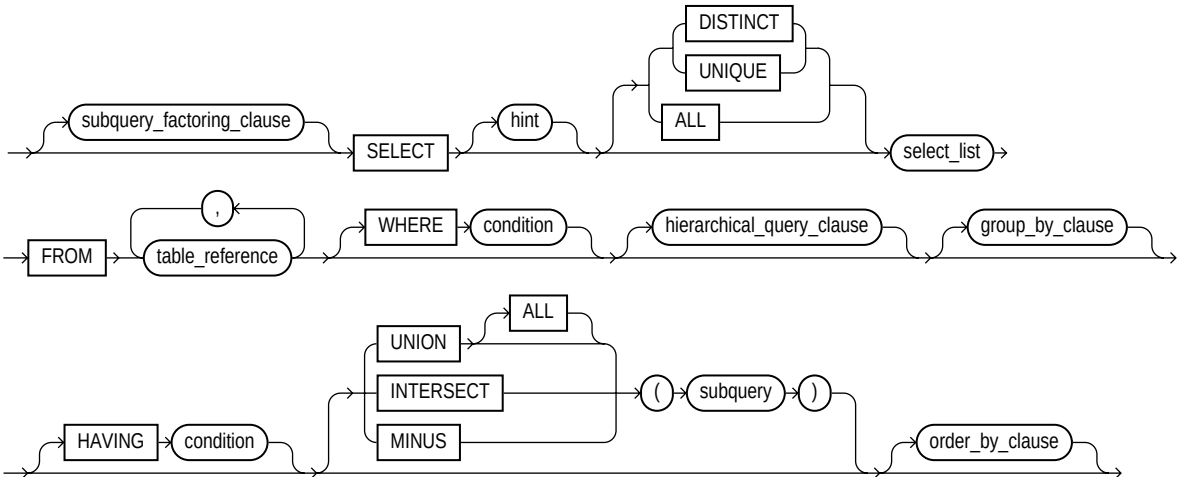
This section describes some types of queries and subqueries and how to use them. The top level of the syntax is shown in this chapter.

> **See Also:** SELECT on page 18-4 for the full syntax of all the clauses and the semantics of the keywords and parameters

select::=



subquery::=



Creating Simple Queries

The list of expressions that appears after the SELECT keyword and before the FROM clause is called the **select list**. Within the select list, you specify one or more columns in the set of rows you want Oracle to return from one or more tables, views, or materialized views. The number of columns, as well as their datatype and length, are determined by the elements of the select list.

If two or more tables have some column names in common, then you must qualify column names with names of tables. Otherwise, fully qualified column names are optional. However, it is always a good idea to qualify table and column references explicitly. Oracle often does less work with fully qualified table and column names.

You can use a column alias, *c_alias*, to label the preceding expression in the select list so that the column is displayed with a new heading. The alias effectively renames the select list item for the duration of the query. The alias can be used in the ORDER BY clause, but not other clauses in the query.

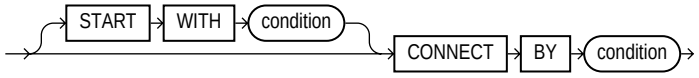
You can use comments in a SELECT statement to pass instructions, or **hints**, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement.

> **See Also:** "Hints" on page 2-92 and *Oracle9i Database Performance Tuning Guide and Reference* for more information on hints

Hierarchical Queries

If a table contains hierarchical data, then you can select rows in a hierarchical order using the hierarchical query clause:

hierarchical_query_clause::=



- START WITH specifies the root row(s) of the hierarchy.
- CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy. In a hierarchical query, one expression in *condition* must be qualified with the PRIOR operator to refer to the parent row. For example,

  ```
  ... PRIOR expr = expr
  or
  ... expr = PRIOR expr
  ```

  If the CONNECT BY *condition* is compound, then only one condition requires the PRIOR operator. For example:

  ```
  CONNECT BY last_name != 'King' AND PRIOR employee_id = manager_id
  ```

  In addition, the CONNECT BY *condition* cannot contain a subquery.

PRIOR is a unary operator and has the same precedence as the unary + and - arithmetic operators. It evaluates the immediately following expression for the parent row of the current row in a hierarchical query.

PRIOR is most commonly used when comparing column values with the equality operator. (The PRIOR keyword can be on either side of the operator.) PRIOR causes Oracle to use the value of the parent row in the column. Operators other than the equal sign (=) are theoretically possible in CONNECT BY clauses. However, the conditions created by these other operators can result in an infinite loop through the possible combinations. In this case Oracle detects the loop at run time and returns an error.

See Also: "Examples" on page 8-5

The manner in which Oracle processes a WHERE clause (if any) in a hierarchical query depends on whether the WHERE clause contains a join:

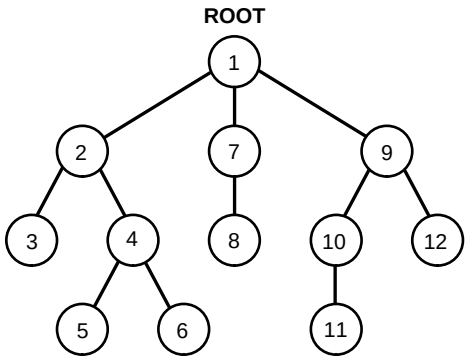
- If the WHERE predicate contains a join, Oracle applies the join predicates *before* doing the CONNECT BY processing.
- If the WHERE clause does not contain a join, Oracle applies all predicates other than the CONNECT BY predicates *after* doing the CONNECT BY processing without affecting the other rows of the hierarchy.

Oracle uses the information from the hierarchical query clause to form the hierarchy using the following steps:

1. Oracle processes the WHERE clause either before or after the CONNECT BY clause depending on whether the WHERE clause contains any join predicates (as described in the preceding bullet list).
2. Oracle selects the root row(s) of the hierarchy—those rows that satisfy the START WITH condition.
3. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the CONNECT BY condition with respect to one of the root rows.
4. Oracle selects successive generations of child rows. Oracle first selects the children of the rows returned in step 3, and then the children of those children, and so on. Oracle always selects children by evaluating the CONNECT BY condition with respect to a current parent row.
5. If the query contains a WHERE clause without a join, then Oracle eliminates all rows from the hierarchy that do not satisfy the condition of the WHERE clause. Oracle evaluates this condition for each row individually, rather than removing all the children of a row that does not satisfy the condition.

6. Oracle returns the rows in the order shown in Figure 8–1. In the diagram, children appear below their parents. For an explanation of hierarchical trees, see Figure 2–1, "Hierarchical Tree" on page 2-87.

Figure 8–1 Hierarchical Queries



To find the children of a parent row, Oracle evaluates the PRIOR expression of the CONNECT BY condition for the parent row and the other expression for each row in the table. Rows for which the condition is true are the children of the parent. The CONNECT BY condition can contain other conditions to further filter the rows selected by the query. The CONNECT BY condition cannot contain a subquery.

If the CONNECT BY condition results in a loop in the hierarchy, then Oracle returns an error. A loop occurs if one row is both the parent (or grandparent or direct ancestor) and a child (or a grandchild or a direct descendent) of another row.

Note: In a hierarchical query, do not specify either ORDER BY or GROUP BY, as they will destroy the hierarchical order of the CONNECT BY results. If you want to order rows of siblings of the same parent, then use the ORDER SIBLINGS BY clause. See *order_by_clause* on page 18-25.

Examples

The following hierarchical query uses the CONNECT BY clause to define the relationship between employees and managers:

```
SELECT employee_id, last_name, manager_id
   FROM employees
```

```
   CONNECT BY PRIOR employee_id = manager_id;

EMPLOYEE_ID LAST_NAME                 MANAGER_ID
----------- ------------------------- ----------
        101 Kochhar                          100
        108 Greenberg                        101
        109 Faviet                           108
        110 Chen                             108
        111 Sciarra                          108
        112 Urman                            108
        113 Popp                             108
        200 Whalen                           101
.
.
.
```

The next example is similar to the preceding example, but uses the LEVEL pseudocolumn to show parent and child rows:

```
SELECT employee_id, last_name, manager_id, LEVEL
   FROM employees
   CONNECT BY PRIOR employee_id = manager_id;

EMPLOYEE_ID LAST_NAME                 MANAGER_ID      LEVEL
----------- ------------------------- ---------- ----------
        101 Kochhar                          100          1
        108 Greenberg                        101          2
        109 Faviet                           108          3
        110 Chen                             108          3
        111 Sciarra                          108          3
        112 Urman                            108          3
        113 Popp                             108          3
.
.
.
```

See Also:

- LEVEL on page 2-86 for a discussion of how the LEVEL pseudocolumn operates in a hierarchical query
- SYS_CONNECT_BY_PATH on page 6-152 for information on retrieving the path of column values from root to node

The UNION [ALL], INTERSECT, MINUS Operators

You can combine multiple queries using the set operators UNION, UNION ALL, INTERSECT, and MINUS. All set operators have equal precedence. If a SQL

statement contains multiple set operators, then Oracle evaluates them from the left to right if no parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, then the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, then the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, then the returned values have datatype VARCHAR2.

Restrictions on set operators:

- The set operators are not valid on columns of type BLOB, CLOB, BFILE, VARRAY, or nested table.
- The UNION, INTERSECT, and MINUS operators are not valid on LONG columns.
- If the select list preceding the set operator contains an expression,then you must provide a column alias for the expression in order to refer to it in the *order_by_clause*.
- You cannot also specify the *for_update_clause* with these set operators.
- You cannot specify the *order_by_clause* in the *subquery* of these operators.
- You cannot use these operators in SELECT statements containing TABLE collection expressions.

> **Note:** To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

The following examples combine the two query results with each of the set operators.

UNION Example The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match datatype (using the TO_CHAR function) when columns do not exist in one or the other table:

```
SELECT location_id, department_name "Department",
   TO_CHAR(NULL) "Warehouse"  FROM departments
   UNION
   SELECT location_id, TO_CHAR(NULL) "Department", warehouse_name
   FROM warehouses;

LOCATION_ID Department            Warehouse
----------- --------------------- --------------------------
       1400 IT
       1400                       Southlake, Texas
       1500 Shipping
       1500                       San Francisco
       1600                       New Jersey
       1700 Accounting
       1700 Administration
       1700 Benefits
       1700 Construction
.
.
.
```

UNION ALL Example The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows:

```
SELECT product_id FROM order_items
UNION
SELECT product_id FROM inventories;

SELECT location_id  FROM locations
UNION ALL
SELECT location_id  FROM departments;
```

A location_id value that appears multiple times in either or both queries (such as '1700') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example The following statement combines the results with the INTERSECT operator, which returns only those rows returned by both queries:

```
SELECT product_id FROM inventories
INTERSECT
SELECT product_id FROM order_items;
```

MINUS Example The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items;
```

Sorting Query Results

Use the ORDER BY clause to order the rows selected by a query. Sorting by position is useful in the following cases:

- To order by a lengthy select list expression, you can specify its position, rather than duplicate the entire expression, in the ORDER BY clause.
- For compound queries (containing set operators UNION, INTERSECT, MINUS, or UNION ALL), the ORDER BY clause must use positions, rather than explicit expressions. Also, the ORDER BY clause can appear only in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The mechanism by which Oracle sorts values for the ORDER BY clause is specified either explicitly by the NLS_SORT initialization parameter or implicitly by the NLS_LANGUAGE initialization parameter. You can change the sort mechanism dynamically from one linguistic sort sequence to another using the ALTER SESSION statement. You can also specify a specific sort sequence for a single query by using the NLSSORT function with the NLS_SORT parameter in the ORDER BY clause.

> **See Also:** *Oracle9i Database Globalization Support Guide* for information on the NLS parameters

Joins

A **join** is a query that combines rows from two or more tables, views, or materialized views. Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, then you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a **join condition**. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which

the join condition evaluates to TRUE. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, in the case of the cost-based optimization approach, statistics for the tables.

In addition to join conditions, the WHERE clause of a join query can also contain other conditions that refer to columns of only one table. These conditions can further restrict the rows returned by the join query.

Note: You cannot specify LOB columns in the WHERE clause if the WHERE clause contains any joins. The use of LOBs in WHERE clauses is also subject to other restrictions. See *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for more information.

Equijoins

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. Depending on the internal algorithm the optimizer chooses to execute the join, the total size of the columns in the equijoin condition in a single table may be limited to the size of a data block minus some overhead. The size of a data block is specified by the initialization parameter DB_BLOCK_SIZE.

See Also: "Using Join Queries: Examples" on page 18-34

Self Joins

A **self join** is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle combines and returns rows of the table that satisfy the join condition.

See Also: "Using Self Joins: Example" on page 18-36

Cartesian Products

If two tables in a join query have no join condition, then Oracle returns their **Cartesian product**. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product. If a query joins three or more tables and you do not specify a join condition for a specific pair, then the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Inner Joins

An **inner join** (sometimes called a "simple join") is a join of two or more tables that returns only those rows that satisfy the join condition.

Outer Joins

An **outer join** extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

- To write a query that performs an outer join of tables A and B and returns all rows from A (a **left outer join**), use the `LEFT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to all columns of B in the join condition in the `WHERE` clause. For all rows in A that have no matching rows in B, Oracle returns null for any select list expressions containing columns of B.
- To write a query that performs an outer join of tables A and B and returns all rows from B (a **right outer join**), use the `RIGHT [OUTER] JOIN` syntax in the `FROM` clause, or apply the outer join operator (+) to all columns of A in the join condition in the `WHERE` clause. For all rows in B that have no matching rows in A, Oracle returns null for any select list expressions containing columns of A.
- To write a query that performs an outer join and returns all rows from A and B, extended with nulls if they do not satisfy the join condition (a **full outer join**), use the `FULL [OUTER] JOIN` syntax in the `FROM` clause.

Oracle Corporation recommends that you use the `FROM` clause `OUTER JOIN` syntax rather than the Oracle join operator. Outer join queries that use the Oracle join operator (+) are subject to the following rules and restrictions, which do not apply to the `FROM` clause join syntax:

- You cannot specify the (+) operator in a query block that also contains `FROM` clause join syntax.

- The (+) operator can appear only in the WHERE clause or, in the context of left-correlation (that is, when specifying the TABLE clause) in the FROM clause, and can be applied only to a column of a table or view.
- If A and B are joined by multiple join conditions, then you must use the (+) operator in all of these conditions. If you do not, then Oracle will return only the rows resulting from a simple join, but without a warning or error to advise you that you do not have the results of an outer join.
- The (+) operator does not produce an outer join if you specify one table in the outer query and the other table in an inner query.
- You cannot use the (+) operator to outer-join a table to itself, although self joins are valid. For example, the following statement is **not** valid:

  ```
  -- The following statement is not valid:
  SELECT employee_id, manager_id
     FROM employees
     WHERE employees.manager_id(+) = employees.employee_id;
  ```

  However, the following self join is valid:

  ```
  SELECT e1.employee_id, e1.manager_id, e2.employee_id
     FROM employees e1, employees e2
     WHERE e1.manager_id(+) = e2.employee_id;
  ```

- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain one or more columns marked with the (+) operator.
- A condition containing the (+) operator cannot be combined with another condition using the OR logical operator.
- A condition cannot use the IN comparison condition to compare a column marked with the (+) operator with an expression.
- A condition cannot compare any column marked with the (+) operator with a subquery.

If the WHERE clause contains a condition that compares a column from table B with a constant, then the (+) operator must be applied to the column so that Oracle returns the rows from table A for which it has generated nulls for this column. Otherwise Oracle will return only the results of a simple join.

In a query that performs outer joins of more than two pairs of tables, a single table can be the null-generated table for only one other table. For this reason, you cannot

apply the (+) operator to columns of B in the join condition for A and B and the join condition for B and C.

See Also: SELECT on page 18-4 for the syntax for an outer join

Using Subqueries

A **subquery** answers multiple-part questions. For example, to determine who works in Taylor's department, you can first use a subquery to determine the department in which Taylor works. You can then answer the original question with the parent `SELECT` statement. A subquery in the `FROM` clause of a `SELECT` statement is also called an **inline view**. A subquery in the `WHERE` clause of a `SELECT` statement is also called a **nested subquery**.

A subquery can contain another subquery. Oracle imposes no limit on the number of subquery levels in the `FROM` clause of the top-level query. You can nest up to 255 levels of subqueries in the `WHERE` clause.

If columns in a subquery have the same name as columns in the containing statement, then you must prefix any reference to the column of the table from the containing statement with the table name or alias. To make your statements easier for you to read, always qualify the columns in a subquery with the name or alias of the table, view, or materialized view.

Oracle performs a **correlated subquery** when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

A correlated subquery answers a multiple-part question whose answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

See Also: "Using Correlated Subqueries: Examples" on page 18-42

Use subqueries for the following purposes:

- To define the set of rows to be inserted into the target table of an `INSERT` or `CREATE TABLE` statement
- To define the set of rows to be included in a view or materialized view in a `CREATE VIEW` or `CREATE MATERIALIZED VIEW` statement

- To define one or more values to be assigned to existing rows in an UPDATE statement
- To provide values for conditions in a WHERE clause, HAVING clause, or START WITH clause of SELECT, UPDATE, and DELETE statements
- To define a table to be operated on by a containing query

  You do this by placing the subquery in the FROM clause of the containing query as you would a table name. You may use subqueries in place of tables in this way as well in INSERT, UPDATE, and DELETE statements.

  Subqueries so used can employ correlation variables, but only those defined within the subquery itself, not outer references. Outer references ("left-correlated subqueries") are allowed only in the FROM clause of a SELECT statement.

  See Also: *table_collection_expression* on page 18-16

  Scalar subqueries, which return a single column value from a single row, are a valid form of expression. You can use scalar subquery expressions in most of the places where *expr* is called for in syntax.

  See Also: "Scalar Subquery Expressions" on page 4-13

Unnesting of Nested Subqueries

Subqueries are "nested" when they appear in the WHERE clause of the parent statement. When Oracle evaluates a statement with a nested subquery, it must evaluate the subquery portion multiple times and may overlook some efficient access paths or joins.

Subquery unnesting unnests and merges the body of the subquery into the body of the statement that contains it, allowing the optimizer to consider them together when evaluating access paths and joins. The optimizer can unnest most subqueries, with some exceptions. Those exceptions include hierarchical subqueries and subqueries that contain a ROWNUM pseudocolumn, one of the set operators, a nested aggregate function, or a correlated reference to a query block that is not the subquery's immediate outer query block.

Assuming no restrictions exist, the optimizer automatically unnests some (but not all) of the following nested subqueries:

- Uncorrelated IN subqueries

- IN and EXISTS correlated subqueries, as long as they do not contain aggregate functions or a GROUP BY clause

You can enable **extended subquery unnesting** by instructing the optimizer to unnest additional types of subqueries:

- You can unnest an uncorrelated NOT IN subquery by specifying the HASH_AJ or MERGE_AJ hint in the subquery.
- You can unnest other subqueries by specifying the UNNEST hint in the subquery.

> **See Also:** Chapter 2, "Basic Elements of Oracle SQL" for information on hints

Selecting from the DUAL Table

DUAL is a table automatically created by Oracle along with the data dictionary. DUAL is in the schema of the user SYS, but is accessible by the name DUAL to all users. It has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with a value 'X'. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table.

> **See Also:** "SQL Functions" on page 6-2 for many examples of selecting a constant value from DUAL

Distributed Queries

Oracle's distributed database management system architecture lets you access data in remote databases using Oracle Net and an Oracle server. You can identify a remote table, view, or materialized view by appending *@dblink* to the end of its name. The *dblink* must be a complete or partial name for a database link to the database containing the remote table, view, or materialized view.

> **See Also:**
>
> - "Referring to Objects in Remote Databases" on page 2-118 for more information on referring to database links
> - *Oracle9i Net Services Administrator's Guide* for information on accessing remote databases

Restrictions on Distributed Queries

Distributed queries are currently subject to the restriction that all tables locked by a FOR UPDATE clause and all tables with LONG columns selected by the query must be located on the same database. For example, the following statement will raise an error:

```
SELECT employees_ny.*
    FROM employees_ny@ny, departments
    WHERE employees_ny.department_id = departments.department_id
    AND departments.department_name = 'ACCOUNTING'
    FOR UPDATE OF employees_ny.salary;
```

The following statement fails because it selects long_column, a LONG value, from the employees_review table on the ny database and locks the employees table on the local database:

```
SELECT employees.employee_id, review.long_column, employees.salary
    FROM employees, employees_review@ny review
    WHERE employees.employee_id = employees_review.employee_id
    FOR UPDATE OF employees.salary;
```

In addition, Oracle currently does not support distributed queries that select user-defined types or object REFs on remote tables.

9

SQL Statements: ALTER CLUSTER to ALTER SEQUENCE

This chapter lists the various types of SQL statements and then describes the first set (in alphabetical order) of SQL statements. The remaining SQL statements appear in alphabetical order in Chapter 10 through Chapter 18.

This chapter contains the following sections:

Types of SQL Statements

The tables in the following sections provide a functional summary of SQL statements and are divided into these categories:

- Data definition language (DDL) statements
- Data manipulation language (DML) statements
- Transaction control statements
- Session control statements
- System control statements

Data Definition Language (DDL) Statements

Data definition language (DDL) statements enable you to perform these tasks:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster
- Establish auditing options
- Add comments to the data dictionary

The `CREATE`, `ALTER`, and `DROP` commands require exclusive access to the specified object. For example, an `ALTER TABLE` statement fails if another user has an open transaction on the specified table.

The `GRANT`, `REVOKE`, `ANALYZE`, `AUDIT`, and `COMMENT` commands do not require exclusive access to the specified object. For example, you can analyze a table while other users are updating the table.

Oracle implicitly commits the current transaction before and after every DDL statement.

Many DDL statements may cause Oracle to recompile or reauthorize schema objects. For information on how Oracle recompiles and reauthorizes schema objects and the circumstances under which a DDL statement would cause this, see *Oracle9i Database Concepts*.

DDL statements are supported by PL/SQL with the use of the `DBMS_SQL` package.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference*

The DDL statements are:

```
ALTER CLUSTER
ALTER DATABASE
ALTER DIMENSION
ALTER FUNCTION
ALTER INDEX
ALTER MATERIALIZED VIEW
ALTER MATERIALIZED VIEW LOG
ALTER PACKAGE
ALTER PROCEDURE
ALTER PROFILE
ALTER RESOURCE COST
ALTER ROLE
ALTER ROLLBACK SEGMENT
ALTER SEQUENCE
ALTER TABLE
ALTER TABLESPACE
ALTER TRIGGER
ALTER TYPE
ALTER USER
ALTER VIEW
ANALYZE
ASSOCIATE STATISTICS
AUDIT
COMMENT
CREATE CLUSTER
CREATE CONTEXT
CREATE CONTROLFILE
CREATE DATABASE
CREATE DATABASE LINK
CREATE DIMENSION
CREATE DIRECTORY
CREATE FUNCTION
CREATE INDEX
CREATE INDEXTYPE
CREATE LIBRARY
CREATE MATERIALIZED VIEW
CREATE MATERIALIZED VIEW LOG
CREATE OPERATOR
CREATE PACKAGE
CREATE PACKAGE BODY
CREATE PFILE
CREATE PROCEDURE
CREATE PROFILE
CREATE ROLE
CREATE ROLLBACK SEGMENT
CREATE SCHEMA
CREATE SEQUENCE
CREATE SPFILE
CREATE SYNONYM
CREATE TABLE
CREATE TABLESPACE
CREATE TEMPORARY TABLESPACE
CREATE TRIGGER
CREATE TYPE
CREATE USER
CREATE VIEW
DISASSOCIATE STATISTICS
DROP CLUSTER
DROP CONTEXT
DROP DATABASE LINK
DROP DIMENSION
DROP DIRECTORY
DROP FUNCTION
DROP INDEX
DROP INDEXTYPE
DROP LIBRARY
DROP MATERIALIZED VIEW
DROP MATERIALIZED VIEW LOG
DROP OPERATOR
DROP PACKAGE
DROP PROCEDURE
DROP PROFILE
DROP ROLE
DROP ROLLBACK SEGMENT
DROP SEQUENCE
DROP SYNONYM
DROP TABLE
DROP TABLESPACE
DROP TRIGGER
DROP TYPE
DROP USER
DROP VIEW
GRANT
NOAUDIT
RENAME
REVOKE
TRUNCATE
```

Data Manipulation Language (DML) Statements

Data manipulation language (DML) statements query and manipulate data in existing schema objects. These statements do not implicitly commit the current transaction. The data manipulation language statements are:

```
CALL
DELETE
EXPLAIN PLAN
INSERT
LOCK TABLE
MERGE
SELECT
UPDATE
```

The `CALL` and `EXPLAIN PLAN` statements are supported in PL/SQL only when executed dynamically. All other DML statements are fully supported in PL/SQL.

Transaction Control Statements

Transaction control statements manage changes made by DML statements. The transaction control statements are:

```
COMMIT
ROLLBACK
SAVEPOINT
SET TRANSACTION
```

All transaction control statements, except certain forms of the `COMMIT` and `ROLLBACK` commands, are supported in PL/SQL. For information on the restrictions, see COMMIT on page 12-72 and ROLLBACK on page 17-100.

Session Control Statements

Session control statements dynamically manage the properties of a user session. These statements do not implicitly commit the current transaction.

PL/SQL does not support session control statements. The session control statements are:

```
ALTER SESSION
SET ROLE
```

System Control Statement

The single system control statement, ALTER SYSTEM, dynamically manages the properties of an Oracle instance. This statement does not implicitly commit the current transaction and is not supported in PL/SQL.

Embedded SQL Statements

Embedded SQL statements place DDL, DML, and transaction control statements within a procedural language program. Embedded SQL is supported by the Oracle precompilers and is documented in the following books:

- *Pro*COBOL Precompiler Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*
- *SQL*Module for Ada Programmer's Guide*

Organization of SQL Statements

All SQL statements in this chapter, as well as in Chapters 10 through 18, are organized into the following sections:

Syntax The syntax diagrams show the keywords and parameters that make up the statement.

> **Caution:** Not all keywords and parameters are valid in all circumstances. Be sure to refer to the "Keywords and Parameters" section of each statement and clause to learn about any restrictions on the syntax.

Purpose The "Purpose" section describes the basic uses of the statement.

Prerequisites The "Prerequisites" section lists privileges you must have and steps that you must take before using the statement. In addition to the prerequisites listed, most statements also require that the database be opened by your instance, unless otherwise noted.

Keywords and Parameters The "Keywords and Parameters" section describes the purpose of each keyword and parameter. (The conventions for keywords and parameters used in this chapter are explained in the Preface of this reference.) Restrictions and usage notes also appear in this section.

Examples The "Examples" section shows how to use various clauses and parameters of the statement.

ALTER CLUSTER

Purpose

Use the ALTER CLUSTER statement to redefine storage and parallelism characteristics of a cluster.

> **Note:** You cannot use this statement to change the number or the name of columns in the cluster key, and you cannot change the tablespace in which the cluster is stored.

See Also:

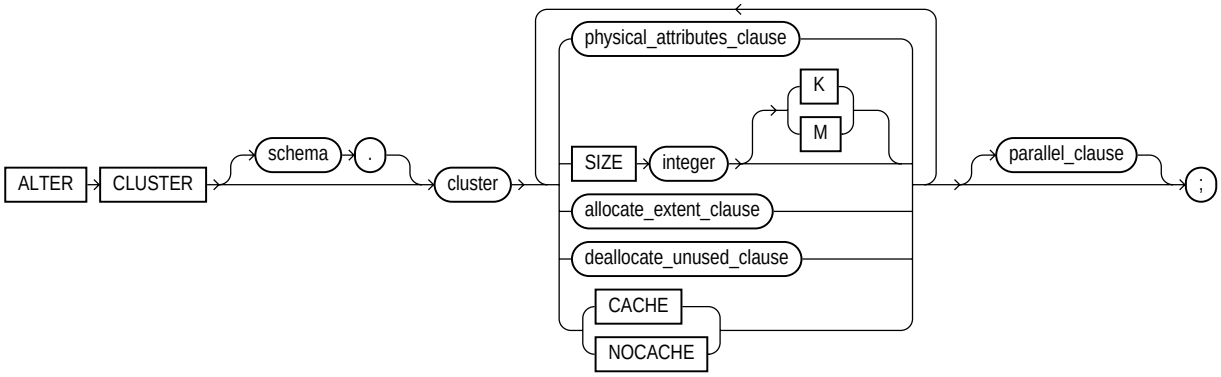
- CREATE CLUSTER on page 13-2 for information on creating a cluster
- DROP CLUSTER on page 16-67 and DROP TABLE on page 17-6 for information on removing tables from a cluster

Prerequisites

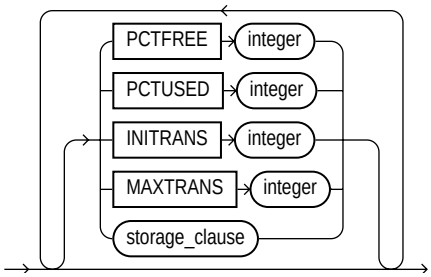
The cluster must be in your own schema or you must have the ALTER ANY CLUSTER system privilege.

Syntax

alter_cluster::=

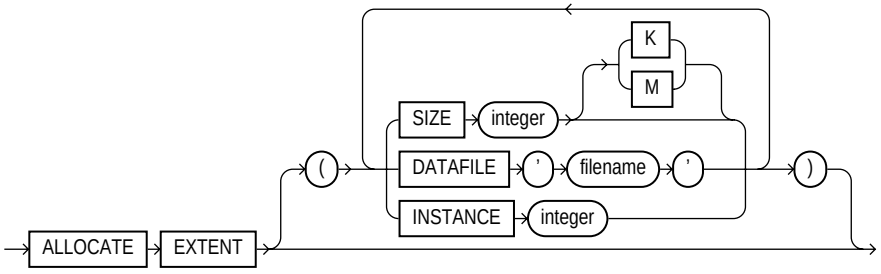


physical_attributes_clause::=

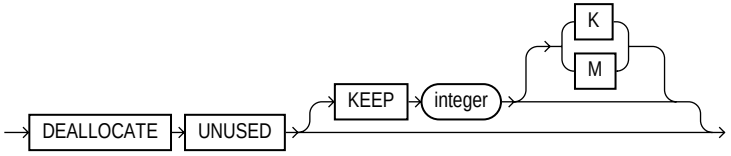


(*storage_clause* on page 7-56)

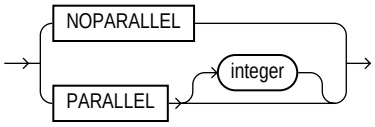
allocate_extent_clause::=



deallocate_unused_clause::=



parallel_clause::=



Keywords and Parameters

schema

Specify the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be altered.

physical_attributes_clause

Use this clause to change the values of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters of the cluster.

Use the STORAGE clause to change the storage characteristics of the cluster.

Restriction on the *physical_attributes_clause***:** You cannot change the values of the storage parameters INITIAL and MINEXTENTS for a cluster.

See Also:

- *physical_attributes_clause* on page 7-52 for a full description of this clause
- *storage_clause* on page 7-56 for a full description of the storage parameters

SIZE *integer*

Use the SIZE clause to specify the number of cluster keys that will be stored in data blocks allocated to the cluster.

Restriction on SIZE: You can change the SIZE parameter only for an indexed cluster, not for a hash cluster.

See Also: CREATE CLUSTER on page 13-2 for a description of the SIZE parameter and "Modifying a Cluster: Example" on page 9-11

allocate_extent_clause

Specify the *allocate_extent_clause* to explicitly allocate a new extent for the cluster.

When you explicitly allocate an extent with this clause, Oracle does not evaluate the cluster's storage parameters and determine a new size for the next extent to be allocated (as it does when you create a table). Therefore, specify SIZE if you do not want Oracle to use a default value.

Restriction on the *allocate_extent_clause*: You can allocate a new extent only for an indexed cluster, not for a hash cluster.

> **See Also:** *allocate_extent_clause* on page 7-2 for a full description of this clause and "Deallocating Unused Space: Example" on page 9-11

deallocate_unused_clause

Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of the cluster and make the freed space available for other segments.

> **See Also:** *deallocate_unused_clause* on page 7-37 for a full description of this clause

CACHE | NOCACHE

CACHE Specify CACHE if you want the blocks retrieved for this cluster to be placed at the *most recently used* end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables.

NOCACHE Specify NOCACHE if you want the blocks retrieved for this cluster to be placed at the *least recently used* end of the LRU list in the buffer cache when a full table scan is performed. This is the default behavior.

parallel_clause

Specify the *parallel_clause* to change the default degree of parallelism for queries and DML on the cluster.

Restriction on the *parallel_clause*: If the tables in *cluster* contain any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *cluster* are executed serially without notification.

> **Note:** The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

Examples

The following examples modify the clusters that were created in the "Examples" section of CREATE CLUSTER on on page 13-9.

Modifying a Cluster: Example The following statement alters the personnel cluster:

```
ALTER CLUSTER personnel
   SIZE 1024
   CACHE;
```

Oracle allocates 1024 bytes for each cluster key value and turns on the cache attribute. Assuming a data block size of 2 kilobytes, future data blocks within this cluster contain 2 cluster keys in each data block, or 2 kilobytes divided by 1024 bytes.

Deallocating Unused Space: Example The following statement deallocates unused space from the language cluster, keeping 30 kilobytes of unused space for future use:

```
ALTER CLUSTER language
    DEALLOCATE UNUSED KEEP 30 K;
```

ALTER DATABASE

Purpose

Use the ALTER DATABASE statement to modify, maintain, or recover an existing database.

See Also:

- *Oracle9i Database Administrator's Guide* for more information on using the ALTER DATABASE statement for database maintenance
- *Oracle9i Database Administrator's Guide*, *Oracle9i User-Managed Backup and Recovery Guide,* and *Oracle9i Recovery Manager User's Guide* for examples of performing media recovery
- *Oracle8i Data Guard Concepts, Administration, and Installation Guide* for additional information on using the ALTER DATABASE statement to maintain standby databases
- CREATE DATABASE on page 13-22 for information on creating a database

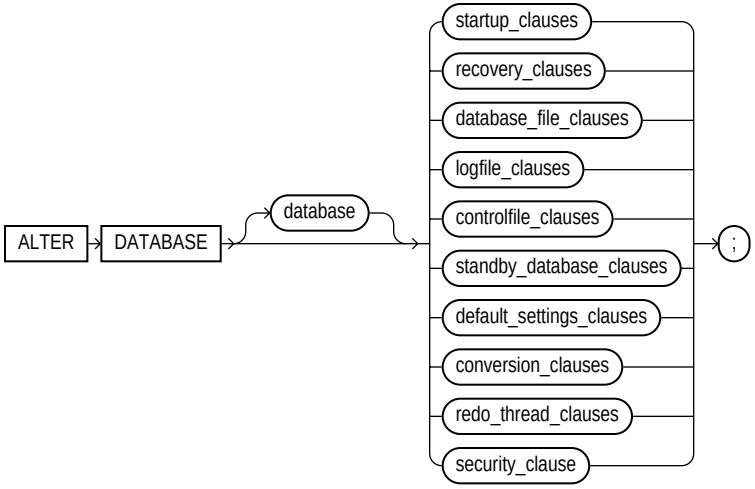
Prerequisites

You must have the ALTER DATABASE system privilege.

To specify the RECOVER clause, you must also have the SYSDBA system privilege.

Syntax

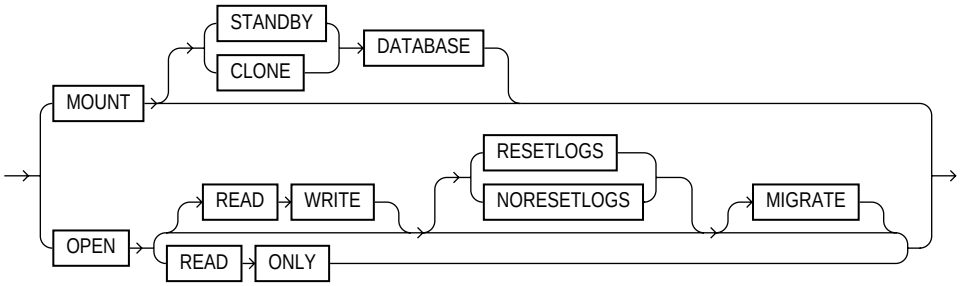
alter_database::=



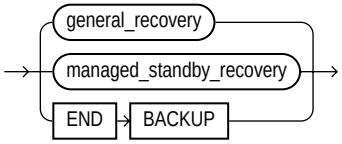
Groups of ALTER DATABASE syntax:

- **startup_clauses::=** on page 9-15
- **recovery_clauses::=** on page 9-15
- **database_file_clauses::=** on page 9-19
- **logfile_clauses::=** on page 9-21
- **controlfile_clauses::=** on page 9-22
- **standby_database_clauses::=** on page 9-23
- **default_settings_clauses::=** on page 9-24
- **conversion_clauses::=** on page 9-24
- **redo_thread_clauses::=** on page 9-24
- **security_clause::=** on page 9-24

startup_clauses::=

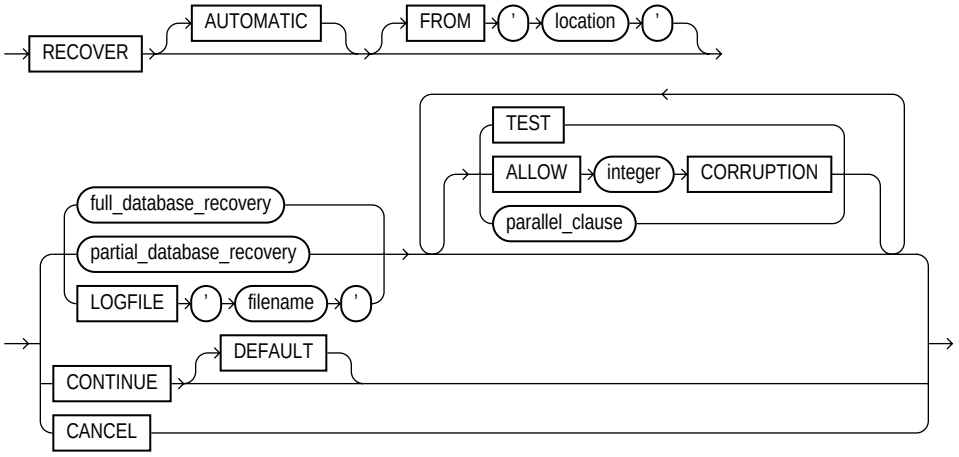


recovery_clauses::=



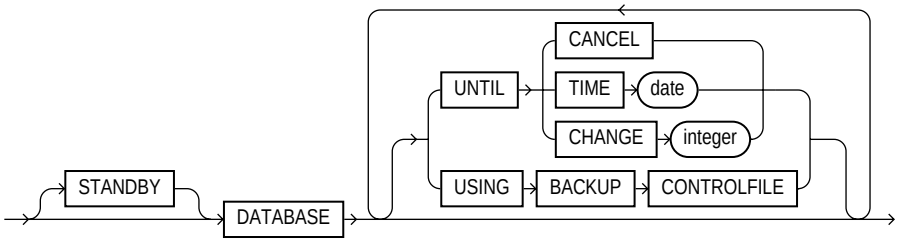
(*general_recovery::=* on page 9-15, *managed_standby_recovery::=* on page 9-17)

general_recovery::=

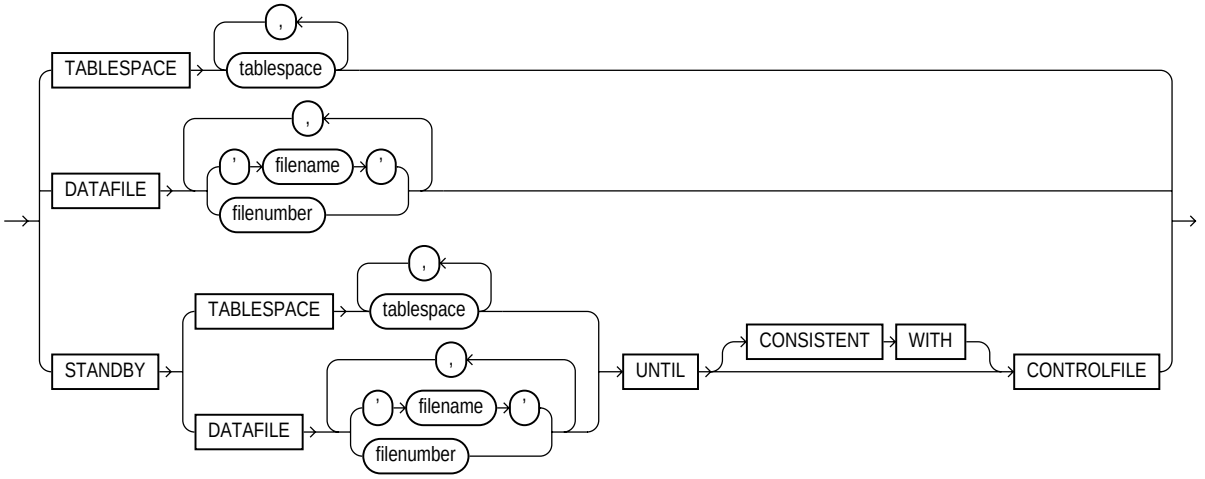


(*full_database_recovery::=* on page 9-16, *partial_database_recovery::=* on page 9-16, *parallel_clause::=* on page 9-16)

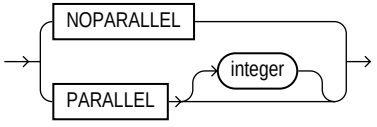
full_database_recovery::=



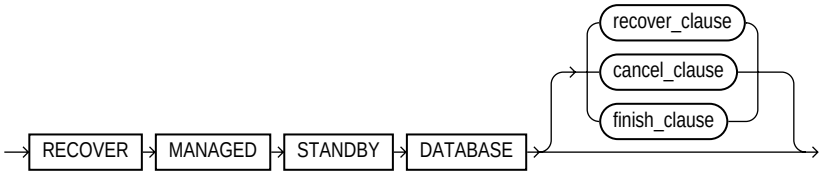
partial_database_recovery::=



parallel_clause::=

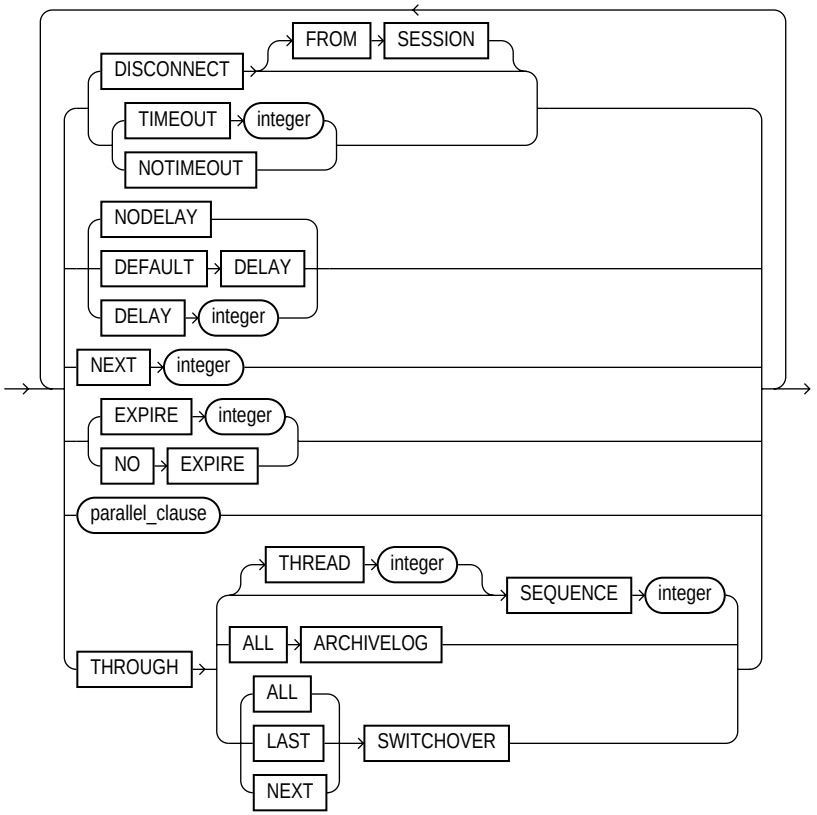


managed_standby_recovery::=



(*recover_clause::=* on page 9-17, *cancel_clause::=* on page 9-18, *finish_clause::=* on page 9-18)

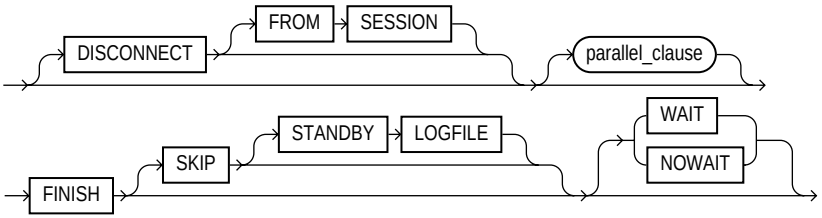
recover_clause::=



(*parallel_clause::=* on page 9-16)

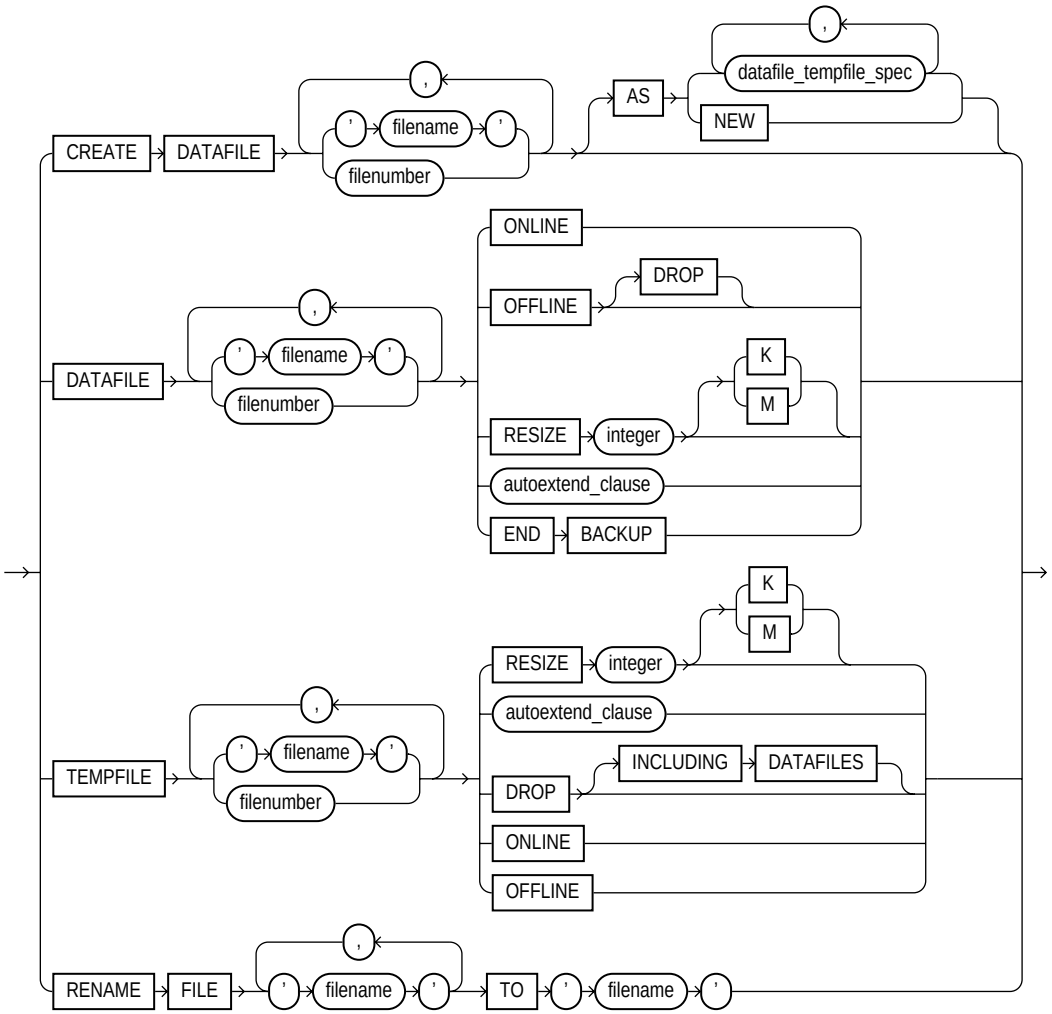
cancel_clause::=

finish_clause::=



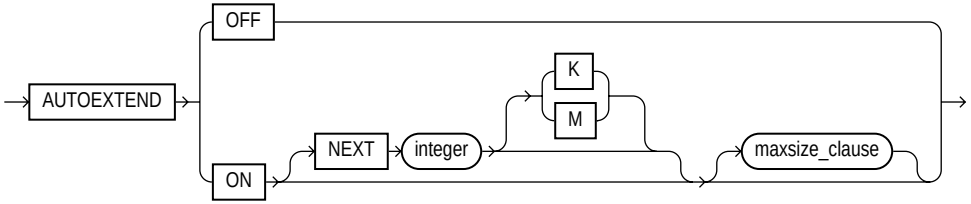
(*parallel_clause::=* on page 9-16)

database_file_clauses::=

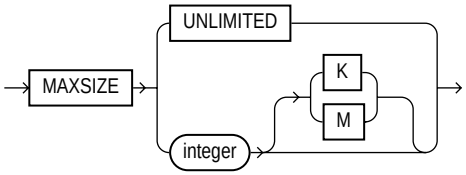


(*datafile_tempfile_spec::=* on page 7-39, *autoextend_clause::=* on page 9-20)

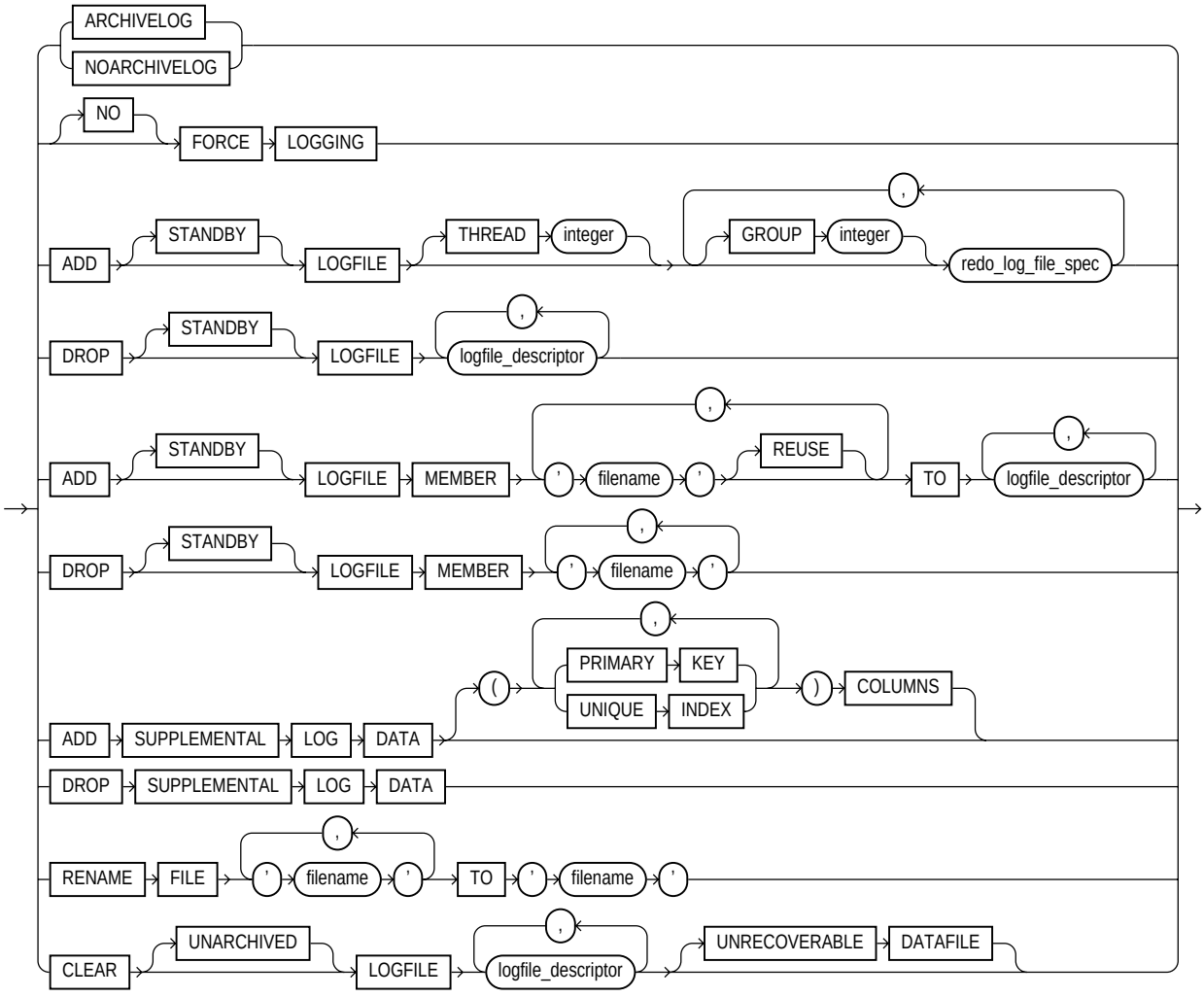
autoextend_clause::=



maxsize_clause::=

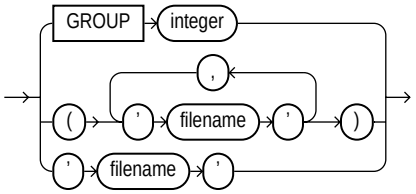


logfile_clauses::=

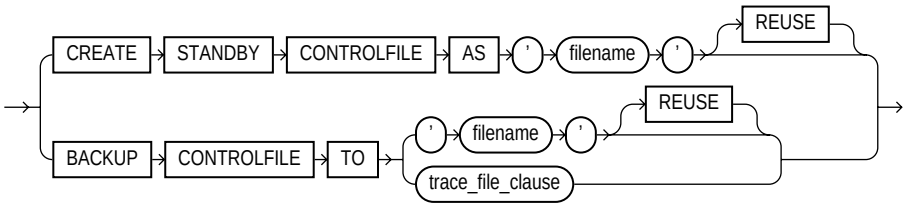


(*redo_log_file_spec::=* on page 7-40)

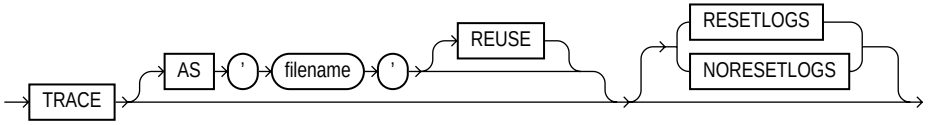
logfile_descriptor::=



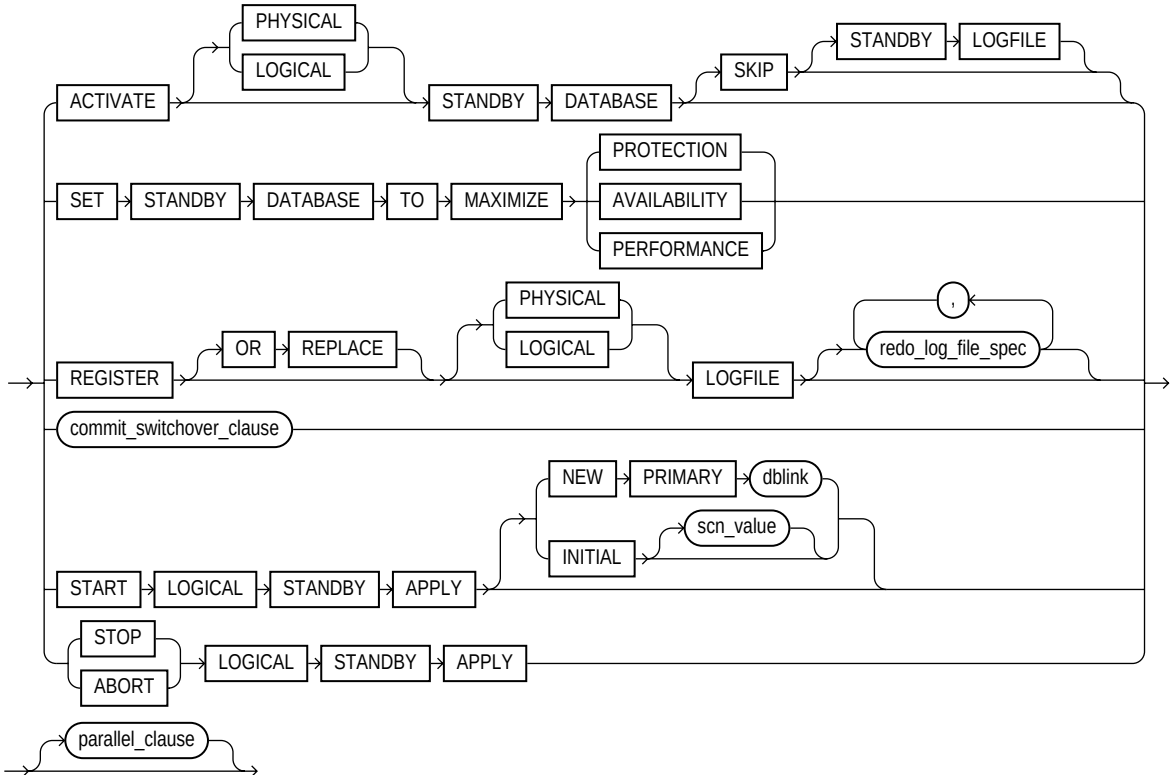
controlfile_clauses::=



trace_file_clause::=

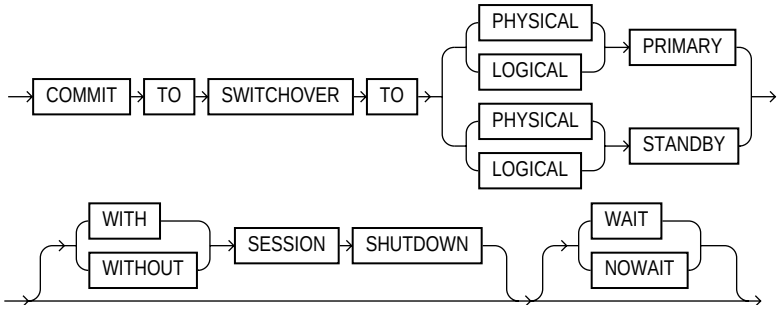


standby_database_clauses::=

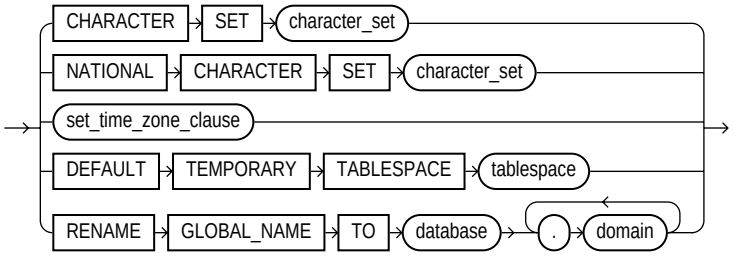


(*redo_log_file_spec::=* on page 7-40)

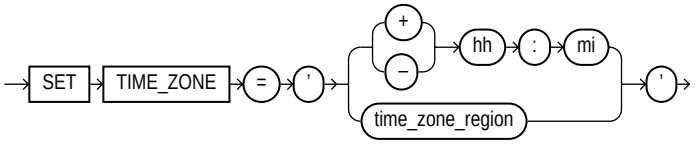
commit_switchover_clause::=



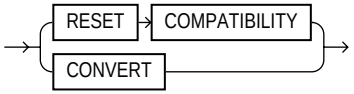
default_settings_clauses::=



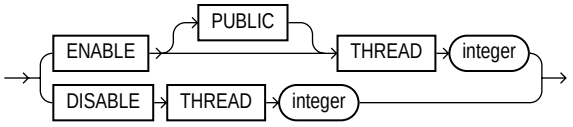
set_time_zone_clause::=



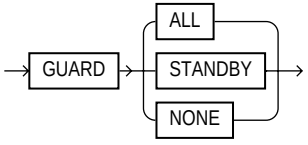
conversion_clauses::=



redo_thread_clauses::=



security_clause::=



Keywords and Parameters

database

Specify the name of the database to be altered. The database name can contain only ASCII characters. If you omit *database*, Oracle alters the database identified by the value of the initialization parameter DB_NAME. You can alter only the database whose control files are specified by the initialization parameter CONTROL_FILES. The database identifier is not related to the Oracle Net database specification.

startup_clauses

The *startup_clauses* let you mount and open the database so that it is accessible to users.

MOUNT Clause

Use the MOUNT clause to mount the database. Do not use this clause when the database is mounted.

MOUNT STANDBY DATABASE Specify MOUNT STANDBY DATABASE to mount a physical standby database. As soon as this statement executes, the standby instance can receive archived redo logs from the primary instance and can archive the logs to the STANDBY_ARCHIVE_DEST location.

See Also: *Oracle9i Data Guard Concepts and Administration*

MOUNT CLONE DATABASE Specify MOUNT CLONE DATABASE to mount the clone database.

See Also: *Oracle9i User-Managed Backup and Recovery Guide*

OPEN Clause

Use the OPEN clause to make the database available for normal use. You must mount the database before you can open it.

If you specify only OPEN, without any other keywords, the default is OPEN READ WRITE NORESETLOGS.

READ WRITE Specify READ WRITE to open the database in read/write mode, allowing users to generate redo logs. This is the default.

See Also: "READ ONLY / READ WRITE: Example" on page 9-53

RESETLOGS Specify RESETLOGS to reset the current log sequence number to 1 and discards any redo information that was not applied during recovery, ensuring that it will never be applied. This effectively discards all changes that are in the redo log, but not in the database.

You must specify RESETLOGS to open the database after performing media recovery with an incomplete recovery using the RECOVER clause or with a backup control file. After opening the database with this clause, you should perform a complete database backup.

NORESETLOGS Specify NORESETLOGS to retain the current state of the log sequence number and redo log files.

Restriction on resetting logs: You can specify RESETLOGS and NORESETLOGS only after performing incomplete media recovery or complete media recovery with a backup control file. In any other case, Oracle uses the NORESETLOGS automatically.

MIGRATE Use the MIGRATE clause only if you are upgrading from Oracle release 7.3.4 to the current release. This clause instructs Oracle to modify system parameters dynamically as required for the upgrade. For upgrade from releases other than 7.3.4, you can use the SQL*Plus STARTUP MIGRATE command.

See Also:

- *Oracle9i Database Migration* for information on the steps required to migrate a database from one release to another
- *SQL*Plus User's Guide and Reference* for information on the SQL*Plus STARTUP command

READ ONLY Specify READ ONLY to restrict users to read-only transactions, preventing them from generating redo logs. You can use this clause to make a physical standby database available for queries even while archive logs are being copied from the primary database site.

Restrictions on the OPEN clause:

- You cannot open a database READ ONLY if it is currently opened READ WRITE by another instance.
- You cannot open a database READ ONLY if it requires recovery.
- You cannot take tablespaces offline while the database is open READ ONLY. However, you can take datafiles offline and online, and you can recover offline datafiles and tablespaces while the database is open READ ONLY.

recovery_clauses

The *recovery_clauses* include post-backup operations.

> **See Also:** *Oracle9i Backup and Recovery Concepts* and *Oracle9i Recovery Manager User's Guide* for information on backing up the database and "Database Recovery: Examples" on page 9-56

general_recovery

The *general_recovery* clause lets you control media recovery for the database or standby database, or for specified tablespaces or files. You can use this clause when your instance has the database mounted, open or closed, and the files involved are not in use.

Restrictions on the *general_recovery_clause*:

- You can recover the entire database only when the database is closed.
- Your instance must have the database mounted in exclusive mode.
- You can recover tablespaces or datafiles when the database is open or closed, if the tablespaces or datafiles to be recovered are offline.
- You cannot perform media recovery if you are connected to Oracle through the Shared Server architecture.

> **Note:** If you do not have special media requirements, Oracle Corporation recommends that you use the SQL*Plus RECOVER command rather than the *general_recovery_clause*.

> **See Also:**
>
> - *Oracle9i User-Managed Backup and Recovery Guide* for more information on media recovery
> - *SQL*Plus User's Guide and Reference* for information on the SQL*Plus RECOVER command

AUTOMATIC

Specify AUTOMATIC if you want Oracle to automatically generate the name of the next archived redo log file needed to continue the recovery operation. If the LOG_ARCHIVE_DEST_*n* parameters are defined, Oracle scans those that are valid and enabled for the first local destination. It uses that destination in conjunction with

LOG_ARCHIVE_FORMAT to generate the target redo log filename. If the LOG_ARCHIVE_DEST_*n* parameters are not defined, Oracle uses the value of the LOG_ARCHIVE_DEST parameter instead.

If the resulting file is found, Oracle applies the redo contained in that file. If the file is not found, Oracle prompts you for a filename, displaying the generated filename as a suggestion.

If you specify neither AUTOMATIC nor LOGFILE, Oracle prompts you for a filename, displaying the generated filename as a suggestion. You can then accept the generated filename or replace it with a fully qualified filename. If you know that the archived filename differs from what Oracle would generate, you can save time by using the LOGFILE clause.

FROM 'location'

Specify FROM *'location'* to indicate the location from which the archived redo log file group is read. The value of *location* must be a fully specified file location following the conventions of your operating system. If you omit this parameter, Oracle assumes that the archived redo log file group is in the location specified by the initialization parameter LOG_ARCHIVE_DEST or LOG_ARCHIVE_DEST_1.

full_database_recovery

The *full_database_recovery* clause lets you recover an entire database.

DATABASE Specify the DATABASE clause to recover the entire database. This is the default. You can use this clause only when the database is closed.

STANDBY DATABASE Specify the STANDBY DATABASE clause to manually recover a physical standby database using the control file and archived redo log files copied from the primary database. The standby database must be mounted but not open.

Note: This clause recovers only online datafiles.

- Use the UNTIL clause to specify the duration of the recovery operation.
  - CANCEL indicates cancel-based recovery. This clause recovers the database until you issue the ALTER DATABASE statement with the RECOVER CANCEL clause.

- TIME indicates time-based recovery. This parameter recovers the database to the time specified by the date. The date must be a character literal in the format 'YYYY-MM-DD:HH24:MI:SS'.
- CHANGE indicates change-based recovery. This parameter recovers the database to a transaction-consistent state immediately before the system change number (SCN) specified by *integer*.

- Specify USING BACKUP CONTROLFILE if you want to use a backup control file instead of the current control file.

partial_database_recovery

The *partial_database_recovery* clause lets you recover individual tablespaces and datafiles.

TABLESPACE Specify the TABLESPACE clause to recover only the specified tablespaces. You can use this clause if the database is open or closed, provided the tablespaces to be recovered are offline.

> **See Also:** "Using Parallel Recovery Processes: Example" on page 9-53

DATAFILE Specify the DATAFILE clause to recover the specified datafiles. You can use this clause when the database is open or closed, provided the datafiles to be recovered are offline.

You can identify the datafile by name or by number. If you identify it by number, then *filenumber* is an integer representing the number found in the FILE# column of the V$DATAFILE dynamic performance view or in the FILE_ID column of the DBA_DATA_FILES data dictionary view.

STANDBY TABLESPACE Specify STANDBY TABLESPACE to reconstruct a lost or damaged tablespace in the standby database using archived redo log files copied from the primary database and a control file.

STANDBY DATAFILE Specify STANDBY DATAFILE to manually reconstruct a lost or damaged datafile in the physical standby database using archived redo log files copied from the primary database and a control file. You can identify the file by name or by number, as described for the DATAFILE clause.

- Specify UNTIL [CONSISTENT WITH] CONTROLFILE if you want the recovery of an old standby datafile or tablespace to use the current standby database control file. However, any redo in advance of the standby controlfile will not be

applied. The keywords CONSISTENT WITH are optional and are provided for semantic clarity.

LOGFILE

Specify the LOGFILE '*filename*' to continue media recovery by applying the specified redo log file.

TEST

Use the TEST clause to conduct a trial recovery. A trial recovery is useful if a normal recovery procedure has encountered some problem. It lets you look ahead into the redo stream to detect possible additional problems. The trial recovery applies redo in a way similar to normal recovery, but it does not write changes to disk, and it rolls back its changes at the end of the trial recovery.

ALLOW ... CORRUPTION

The ALLOW *integer* CORRUPTION clause lets you specify, in the event of logfile corruption, the number of corrupt blocks that can be tolerated while allowing recovery to proceed.

When you use this clause during trial recovery (that is, in conjunction with the TEST clause), *integer* can exceed 1. When using this clause during normal recovery, *integer* cannot exceed 1.

See Also:

- *Oracle9i User-Managed Backup and Recovery Guide* for information on database recovery in general
- *Oracle9i Data Guard Concepts and Administration* for information on managed recovery of standby databases

parallel_clause

Use the PARALLEL clause to specify whether the recovery of media will be parallelized.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

CONTINUE

Specify CONTINUE to continue multi-instance recovery after it has been interrupted to disable a thread.

Specify CONTINUE DEFAULT to continue recovery using the redo log file that Oracle would automatically generate if no other logfile were specified. This clause is equivalent to specifying AUTOMATIC, except that Oracle does not prompt for a filename.

CANCEL

Specify CANCEL to terminate cancel-based recovery.

managed_standby_recovery

The *managed_standby_recovery* clause applies to physical standby only. Use it to specify managed standby recovery mode. This mode assumes that the managed standby database is an active component of an overall standby database architecture. A primary database actively archives its redo log files to the standby site. As these archived redo logs arrive at the standby site, they become available for use by a managed standby recovery operation. Managed standby recovery is restricted to media recovery. You can use this clause when your instance has the database mounted, open or closed, and the files involved are not in use.

Restrictions on the *managed_standby_recovery* **clause:** The same restrictions apply as are listed under *general_recovery* on page 9-27.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for more information on the parameters of this clause "Recovering a Managed Standby Database: Examples" on page 9-56

DISCONNECT Specify DISCONNECT to indicate that the managed redo process (MRP), an Oracle background process, should apply archived redo files as a detached background process. Doing so leaves the current session available for other tasks. (The FROM SESSION keywords are optional and are provided for semantic clarity.)

Restrictions on DISCONNECT:

- You can specify DISCONNECT only when you are initiating managed standby recovery. You cannot specify it after the operation has started.
- You cannot specify both TIMEOUT and DISCONNECT [FROM SESSION]. TIMEOUT applies only to foreground recovery operations, whereas the DISCONNECT clause initiates background recovery operations.

TIMEOUT *integer* Specify in minutes the wait period of the managed recovery operation. The recovery process waits for *integer* minutes for a requested archived log redo to be available for writing to the managed standby database. If the redo log file does not become available within that time, the recovery process terminates with an error message. You can then issue the statement again to return to managed standby recovery mode.

If you omit TIMEOUT or if you specify NOTIMEOUT, the database remains in managed standby recovery mode until you reissue the statement with the RECOVER CANCEL clause or until instance shutdown or failure.

Restrictions on TIMEOUT:

- If you specify TIMEOUT, you cannot also specify FINISH.
- You cannot specify both TIMEOUT and DISCONNECT [FROM SESSION]. TIMEOUT applies only to foreground recovery operations, whereas the DISCONNECT clause initiates background recovery operations.

NODELAY | DEFAULT DELAY | DELAY *integer* Specify DELAY to instruct Oracle to wait the specified interval (in minutes) before applying the archived redo logs. The delay interval begins after the archived redo logs have been selected for recovery.

- Specify NODELAY if the need arises to apply a delayed archivelog immediately on the standby database.

- Specify DEFAULT DELAY to revert to the number of minutes specified in the LOG_ARCHIVE_DEST_*n* initialization parameter on the primary database.

Both of these parameters override any setting of DELAY in the LOG_ARCHIVE_DEST_*n* parameter on the primary database. If you specify neither of these parameters, application of the archivelog is delayed according to the LOG_ARCHIVE_DEST_*n* setting. If DELAY was not specified in that parameter, the archivelog is applied immediately.

Restrictions on DELAY:

- You cannot specify both NODELAY and DELAY.
- If you specify DELAY, you cannot also specify FINISH.

> **See Also:** *Oracle9i Database Reference* for detailed information on the LOG_ARCHIVE_DEST_*n* parameter

NEXT *integer* Use the NEXT parameter to apply the specified number of archived redo logs as soon as possible after they have been archived. This parameter temporarily overrides any delay setting in the LOG_ARCHIVE_DEST_*n* initialization parameter on the primary database and over any DELAY values specified in an earlier ALTER DATABASE ... *managed_standby_recovery* statement. Once the *integer* archived redo logs are processed, any such delay again takes effect.

Restriction on NEXT: If you specify NEXT, you cannot also specify FINISH.

> **See Also:** *Oracle9i Database Reference* for detailed information on the LOG_ARCHIVE_DEST_*n* parameter

EXPIRE *integer* Specify the number of minutes from the current time after which the managed recovery operation terminates automatically. The process may actually expire after the interval specified, because Oracle will finish processing any archived redo log that is being processed at the expiration time.

Specify NOEXPIRE to disable a previously specified EXPIRE option.

Expiration is always relative to the time the current statement is issued rather than to the start time of the managed recovery process. To terminate an existing managed recovery operation, use the CANCEL parameter.

Restriction on EXPIRE: If you specify EXPIRE, you cannot also specify FINISH.

THROUGH Clause Use this clause to instruct Oracle when to terminate managed recovery.

- THROUGH ... SEQUENCE: Specify this clause if you want Oracle to terminate managed recovery based on thread number and sequence number of an archivelog. Once the corresponding archivelog has been applied, managed recovery terminates. If you omit the THREAD clause, Oracle assumes thread 1.
- THROUGH ALL ARCHIVELOG: Specify this clause if you want Oracle to continue the managed standby process until all archivelogs have been recovered. You can use this statement to override an earlier statement that specified THROUGH ... SEQUENCE. If you omit the THROUGH clause entirely, this is the default.
- THROUGH ... SWITCHOVER: The managed standby recovery process normally stops when it encounters a switchover operation, because these operations produce an "end-of-redo archival" indicator. This clause is useful if you have more than one standby database, all but one of which will remain in the standby role after the switchover. This clause keeps the managed standby recovery process operational. It lets these "secondary" standby databases wait to receive the redo stream from the new primary database, rather than stopping the recovery process and then starting it again after the new primary database is activated.
  - Specify ALL to keep managed standby recovery operational through all switchover operations.
  - Specify LAST to cancel managed standby recovery operations after the final end-of-redo archival indicator.
  - Specify NEXT to cancel managed standby recovery after recovering the next end-of-redo archival indicator encountered. This is the default.

CANCEL Specify CANCEL to terminate the managed standby recovery operation after applying all the redo in the current archived redo file. If you specify only the CANCEL keyword, session control returns when the recovery process actually terminates.

- Specify CANCEL IMMEDIATE to terminate the managed recovery operation after applying all the redo in the current archived redo file or after the next redo log file read, whichever comes first. Session control returns when the recovery process actually terminates.

  Restriction on CANCEL IMMEDIATE: The CANCEL IMMEDIATE clause cannot be issued from the same session that issued the RECOVER MANAGED STANDBY DATABASE statement.
- CANCEL IMMEDIATE NOWAIT is the same as CANCEL IMMEDIATE except that session control returns immediately, not after the recovery process terminates.

- CANCEL NOWAIT terminates the managed recovery operation after the next redo log file read and returns session control immediately.

FINISH The FINISH clause applies only to physical standby databases. Specify FINISH to recover the current standby online redo logfiles of the standby database. Use this clause only in the event of the failure of the primary database, when the logwriter (LGWR) process has been transmitting redo to the standby current logs. This clause overrides any delay intervals specified for the archivelogs, so that Oracle applies the logs immediately.

After the FINISH operation, you must open the standby database as the primary database.

Specify NOWAIT to have control returned immediately rather than after the recovery process is complete.

Restrictions on FINISH: You cannot specify FINISH if you have also specified TIMEOUT, DELAY, EXPIRE, or NEXT.

parallel_clause Use the *parallel_clause* to indicate whether Oracle should parallelize the managed recovery processes. If you specify NOPARALLEL or omit this clause entirely, Oracle performs the managed standby recovery operation serially.

See Also:

- *parallel_clause* on page 7-49 for more information on this clause
- *Oracle9i Data Guard Concepts and Administration* for guidelines on determining whether parallel managed standby recovery will result in performance gains

END BACKUP Clause

Specify END BACKUP to take out of online backup mode any datafiles in the database currently in online backup mode. The database must be mounted but not open when you perform this operation.

You can end online ("hot") backup operations in three ways. During normal operation, you can take a tablespace out of online backup mode using the ALTER TABLESPACE ... END BACKUP statement. Doing so avoids the increased overhead of leaving the tablespace in online backup mode.

After a system failure, instance failure, or SHUTDOWN ABORT operation, Oracle does not know whether the files in online backup mode match the files at the time the system crashed. If you know the files are consistent, you can take either individual datafiles or all datafiles out of online backup mode. Doing so avoids media recovery of the files upon startup.

- To take an individual datafile out of online backup mode, use the ALTER DATABASE DATAFILE ... END BACKUP statement. See *database_file_clauses* on page 9-36.
- To take all datafiles in a tablespace out of online backup mode, use an ALTER TABLESPACE ... END BACKUP statement.

> **See Also:** ALTER TABLESPACE on page 11-101 for information on ending online tablespace backup

database_file_clauses

The *database_file_clauses* let you modify datafiles and tempfiles. You can use any of the following clauses when your instance has the database mounted, open or closed, and the files involved are not in use.

CREATE DATAFILE

Use the CREATE DATAFILE clause to create a new empty datafile in place of an old one. You can use this clause to re-create a datafile that was lost with no backup. The *filename* or *filenumber* must identify a file that is or was once part of the database. If you identify the file by number, then *filenumber* is an integer representing the number found in the FILE# column of the V$DATAFILE dynamic performance view or in the FILE_ID column of the DBA_DATA_FILES data dictionary view.

- Specify AS NEW to create an Oracle-managed datafile with a system-generated filename, the same size as the file being replaced, in the default file system location for datafiles.
- Specify AS *datafile_tempfile_spec* to assign a filename (and optional size) for the new datafile.

If the original file (*filename* or *filenumber*) is an existing Oracle-managed datafile, then Oracle attempts to delete the original file after creating the new file. If the original file is an existing user-managed datafile, Oracle does not attempt to delete the original file.

If you omit the AS clause entirely, Oracle creates the new file with the same name and size as the file specified by *filename* or *filenumber*.

During recovery, all archived redo logs written to since the original datafile was created must be applied to the new, empty version of the lost datafile.

Oracle creates the new file in the same state as the old file when it was created. You must perform media recovery on the new file to return it to the state of the old file at the time it was lost.

Restrictions on creating new datafiles:

- You cannot create a new file based on the first datafile of the SYSTEM tablespace.
- You cannot specify the *autoextend_clause* of *datafile_tempfile_spec* in this CREATE DATAFILE clause.

> **See Also:** *file_specification* on page 7-39 for a full description of the file specification (*datafile_tempfile_spec*) and "Creating a New Datafile: Example" on page 9-55

DATAFILE Clauses

The DATAFILE clauses let you manipulate a file that you identify by name or by number. If you identify it by number, then *filenumber* is an integer representing the number found in the FILE# column of the V$DATAFILE dynamic performance view or in the FILE_ID column of the DBA_DATA_FILES data dictionary view. The DATAFILE clauses affect your database files as follows:

ONLINE Specify ONLINE to bring the datafile online.

OFFLINE Specify OFFLINE to take the datafile offline. If the database is open, you must perform media recovery on the datafile before bringing it back online, because a checkpoint is not performed on the datafile before it is taken offline.

DROP If the database is in NOARCHIVELOG mode, you must specify the DROP clause to take a datafile offline. However, the DROP clause does not remove the datafile from the database. To do that, you must drop the tablespace in which the datafile resides. Until you do so, the datafile remains in the data dictionary with the status RECOVER or OFFLINE.

If the database is in ARCHIVELOG mode, Oracle ignores the DROP keyword.

RESIZE Specify RESIZE if you want Oracle to attempt to increase or decrease the size of the datafile to the specified absolute size in bytes. Use K or M to specify this size in kilobytes or megabytes. There is no default, so you must specify a size.

If sufficient disk space is not available for the increased size, or if the file contains data beyond the specified decreased size, Oracle returns an error.

> **See Also:** "Resizing a Datafile: Example" on page 9-55

END BACKUP Specify END BACKUP to take the datafile out of online backup mode. The END BACKUP clause is described more fully at the top level of the syntax of ALTER DATABASE. See "END BACKUP Clause" on page 9-35.

TEMPFILE Clause

Use the TEMPFILE clause to resize your temporary datafile or specify the *autoextend_clause*, with the same effect as with a permanent datafile. You can identify the tempfile by name or by number. If you identify it by number, then *filenumber* is an integer representing the number found in the FILE# column of the V$TEMPFILE dynamic performance view.

Note: On some operating systems, Oracle does not allocate space for the tempfile until the tempfile blocks are actually accessed. This delay in space allocation results in faster creation and resizing of tempfiles, but it requires that sufficient disk space is available when the tempfiles are later used. Please refer to the Oracle documentation for your operating system to determine whether Oracle allocates tempfile space in this way on your system.

Restriction on the TEMPFILE clause: You cannot specify TEMPFILE unless the database is open.

DROP Specify DROP to drop *tempfile* from the database. The tablespace remains.

If you specify INCLUDING DATAFILES, Oracle also deletes the associated operating system files and writes a message to the alert log for each such deleted file.

autoextend_clause

Use the *autoextend_clause* to enable or disable the automatic extension of a new datafile or tempfile.

> **See Also:** *file_specification* on page 7-39 for information about the *autoextend_clause*

RENAME FILE Clause

Use the RENAME FILE clause to rename datafiles, tempfiles, or redo log file members. You must create each filename using the conventions for filenames on your operating system before specifying this clause.

- To use this clause for datafiles and tempfiles, the database must be mounted. The database can also be open, but the datafile or tempfile being renamed must be offline.
- To use this clause for logfiles, the database must be mounted but not open.

This clause renames only files in the control file. It does not actually rename them on your operating system. The operating system files continue to exist, but Oracle no longer uses them. If the old files were Oracle managed, Oracle drops the old operating system file after this statement executes, because the control file no longer points to them as datafiles, tempfiles, or redo log files.

See Also: "Renaming a Log File Member: Example" on page 9-54

logfile_clauses

The logfile clauses let you add, drop, or modify log files.

ARCHIVELOG | NOARCHIVELOG

Use the ARCHIVELOG clause and NOARCHIVELOG clause only if your instance has the database mounted but not open, with Real Application Clusters disabled.

ARCHIVELOG Specify ARCHIVELOG if you want the contents of a redo log file group to be archived before the group can be reused. This mode prepares for the possibility of media recovery. Use this clause only after shutting down your instance normally, or immediately with no errors, and then restarting it and mounting the database with Real Application Clusters disabled.

NOARCHIVELOG Specify NOARCHIVELOG if you do not want the contents of a redo log file group to be archived so that the group can be reused. This mode does not prepare for recovery after media failure.

[NO] FORCE LOGGING

Use this clause to put the database into or take the database out of FORCE LOGGING mode. The database must be mounted or open.

In FORCE LOGGING mode, Oracle will log all changes in the database except for changes in temporary tablespaces and temporary segments. This setting takes

precedence over and is independent of any NOLOGGING or FORCE LOGGING settings you specify for individual tablespaces and any NOLOGGING settings you specify for individual database objects.

If you specify FORCE LOGGING, Oracle waits for all ongoing unlogged operations to finish.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on when to use FORCE LOGGING mode

ADD [STANDBY] LOGFILE Clause

Use the ADD LOGFILE clause to add one or more redo log file groups to the specified thread, making them available to the instance assigned the thread. If you specify STANDBY, the redo log file created is for use by physical standby databases only.

To learn whether a logfile has been designated for online or standby database use, query the TYPE column of the V$LOGFILE dynamic performance view.

> **See Also:** "Adding Redo Log File Groups: Examples" on page 9-53

THREAD The THREAD clause is applicable only if you are using Oracle with the Real Application Clusters option in parallel mode. *integer* is the thread number. The number of threads you can create is limited by the value of the MAXINSTANCES parameter specified in the CREATE DATABASE statement.

If you omit THREAD, the redo log file group is added to the thread assigned to your instance.

GROUP The GROUP clause uniquely identifies the redo log file group among all groups in all threads and can range from 1 to the MAXLOGFILES value. You cannot add multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

redo_log_file_spec Each *redo_log_file_spec* specifies a redo log file group containing one or more members (that is, one or more copies).

See Also:

- *file_specification* on page 7-39
- *Oracle9i Database Reference* for information on dynamic performance views

DROP LOGFILE Clause

Use the DROP LOGFILE clause to drop all members of a redo log file group. Specify a redo log file group as indicated for the ADD LOGFILE MEMBER clause.

- To drop the current log file group, you must first issue an ALTER SYSTEM SWITCH LOGFILE statement.
- You cannot drop a redo log file group if it needs archiving.
- You cannot drop a redo log file group if doing so would cause the redo thread to contain less than two redo log file groups.

See Also: ALTER SYSTEM on page 10-22 and "Dropping Log File Members: Example" on page 9-54

ADD [STANDBY] LOGFILE MEMBER Clause

Use the ADD LOGFILE MEMBER clause to add new members to existing redo log file groups. Each new member is specified by *'filename'*. If the file already exists, it must be the same size as the other group members, and you must specify REUSE. If the file does not exist, Oracle creates a file of the correct size. You cannot add a member to a group if all of the group's members have been lost through media failure.

You can specify STANDBY for symmetry, to indicate that the logfile member is for use only by a physical standby database. However, this keyword is not required. If group *integer* was added for standby database use, all of its members will be used only for standby databases as well.

You can specify an existing redo log file group in one of two ways:

GROUP *integer* Specify the value of the GROUP parameter that identifies the redo log file group.

***filename*(s)** List all members of the redo log file group. You must fully specify each filename according to the conventions of your operating system.

See Also: "Adding Redo Log File Group Members: Example" on page 9-54

DROP LOGFILE MEMBER Clause

Use the DROP LOGFILE MEMBER clause to drop one or more redo log file members. Each *'filename'* must fully specify a member using the conventions for filenames on your operating system.

- To drop a log file in the current log, you must first issue an ALTER SYSTEM SWITCH LOGFILE statement.

  See Also: ALTER SYSTEM on page 10-22

- You cannot use this clause to drop all members of a redo log file group that contains valid data. To perform that operation, use the DROP LOGFILE clause.

  See Also: "Dropping Log File Members: Example" on page 9-54

ADD SUPPLEMENTAL LOG DATA Clause

Specify the ADD SUPPLEMENTAL LOG DATA clause to place additional column data into the log stream any time an update operation is performed. These four keywords alone enable **minimal supplemental logging**, which is not enabled by default.

Minimal supplemental logging ensures that Logminer (and any products building on Logminer technology) will have sufficient information to support chained rows and various storage arrangements such as cluster tables.

If supplemental log data will be the source of change in another database, such as a logical standby, the log data must also uniquely identify each row updated. In this case, you should enable **identification key ("full") supplemental** logging by specifying PRIMARY KEY COLUMNS and UNIQUE KEY COLUMNS.

PRIMARY KEY COLUMNS When you specify PRIMARY KEY COLUMNS, Oracle ensures, for all tables with a primary key, that all columns of the primary key are placed into the redo log whenever an update operation is performed. If no primary key is defined, Oracle places into the redo log a set of columns that uniquely identifies the row. This set may include all columns with a fixed-length maximum size.

UNIQUE INDEX COLUMNS When you specify UNIQUE INDEX COLUMNS, Oracle ensures, for all tables with a unique key, that if any unique key columns are

modified, all other columns belonging to the unique key are also placed into the redo log.

Note: You can issue this statement when the database is open. However, Oracle will invalidate all DML cursors in the cursor cache, which will have an effect on performance until the cache is repopulated.

See Also: *Oracle9i Data Guard Concepts and Administration* for information on supplemental logging

DROP SUPPLEMENTAL LOG DATA Clause

Use the DROP SUPPLEMENTAL LOG DATA clause to instruct Oracle to stop placing additional log information into the redo log stream whenever an update operation occurs. This statement terminates the effect of a previous ADD SUPPLEMENTAL LOG DATA statement.

See Also: *Oracle9i Data Guard Concepts and Administration* for information on supplemental logging

CLEAR LOGFILE Clause

Use the CLEAR LOGFILE clause to reinitialize an online redo log, optionally without archiving the redo log. CLEAR LOGFILE is similar to adding and dropping a redo log, except that the statement may be issued even if there are only two logs for the thread and also may be issued for the current redo log of a closed thread.

- You must specify UNARCHIVED if you want to reuse a redo log that was not archived.

Caution: Specifying UNARCHIVED makes backups unusable if the redo log is needed for recovery.

- You must specify UNRECOVERABLE DATAFILE if you have taken the datafile offline with the database in ARCHIVELOG mode (that is, you specified ALTER DATABASE ... DATAFILE OFFLINE without the DROP keyword), and if the unarchived log to be cleared is needed to recover the datafile before bringing it back online. In this case, you must drop the datafile and the entire tablespace once the CLEAR LOGFILE statement completes.

Do not use CLEAR LOGFILE to clear a log needed for media recovery. If it is necessary to clear a log containing redo after the database checkpoint, you must first perform incomplete media recovery. The current redo log of an open thread can be cleared. The current log of a closed thread can be cleared by switching logs in the closed thread.

If the CLEAR LOGFILE statement is interrupted by a system or instance failure, then the database may hang. If this occurs, reissue the statement after the database is restarted. If the failure occurred because of I/O errors accessing one member of a log group, then that member can be dropped and other members added.

See Also: "Clearing a Log File: Example" on page 9-56

controlfile_clauses

The *controlfile_clauses* let you create or back up a control file.

CREATE STANDBY CONTROLFILE Clause

The CREATE STANDBY CONTROLFILE clause applies only to physical standby databases. Use this clause to create a control file to be used to maintain a physical standby database. If the file already exists, you must specify REUSE.

See Also: *Oracle9i Data Guard Concepts and Administration*

BACKUP CONTROLFILE Clause

Use the BACKUP CONTROLFILE clause to back up the current control file. The database must be open or mounted when you specify this clause.

TO '*filename*' Specify the file to which the control file is backed up. You must fully specify the *filename* using the conventions for your operating system. If the specified file already exists, you must specify REUSE.

TO TRACE Specify TO TRACE if you want Oracle to write SQL statements to a trace file rather than making a physical backup of the control file. You can use SQL statements written to the trace file to start up the database, re-create the control file, and recover and open the database appropriately, based on the created control file.

You can copy the statements from the trace file into a script file, edit the statements as necessary, and use the script if all copies of the control file are lost (or to change the size of the control file).

- Specify AS *filename* if you want Oracle to place the script into a file called *filename* rather than into the standard trace file.
- Specify REUSE to allow Oracle to overwrite any existing file called *filename*.
- RESETLOGS indicates that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN RESETLOGS. This setting is valid only if the online logs are unavailable.
- NORESETLOGS indicates that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN NORESETLOGS. This setting is valid only if all the online logs are available.

If you cannot predict the future state of the online logs, specify neither RESETLOGS nor NORESETLOGS. In this case, Oracle puts both versions of the script into the trace file, and you can choose which version is appropriate when the script becomes necessary.

standby_database_clauses

Use these clauses to activate the standby database or to specify whether it is in protected or unprotected mode.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for descriptions of physical and logical the standby database and for information on maintaining and using standby databases

ACTIVATE STANDBY DATABASE Clause

The ACTIVATE STANDBY DATABASE clause changes the state of a standby database to an active database and prepares it to become the primary database. The database must be mounted before you can specify this clause.

PHYSICAL Specify PHYSICAL to activate a physical standby database. This is the default.

LOGICAL Specify LOGICAL to activate a logical standby database. If you have more than one logical standby database, you should first ensure that the same log data is available on all the standby systems.

SKIP [STANDBY LOGFILE] This clause applies only to physical standby databases. Use this clause to force the operation to proceed even if standby redo logfiles contain data that could be recovered using the RECOVER MANAGED STANDBY DATABASE FINISH command.

> **Note:** Oracle Corporation recommends that you always use the RECOVER MANAGED STANDBY DATABASE FINISH statement for physical standby even if you do not use standby redo logfiles. Use the SKIP clause only if it is acceptable to discard the contents of the standby redo log.

SET STANDBY [DATABASE] Clause

Use this clause to specify the level of protection for the data in your database environment. You specify this clause from the primary database, which must be mounted but not open.

> **Note:** The PROTECTED and UNPROTECTED keywords have been replaced for clarity but are still supported. PROTECTED is equivalent to TO MAXIMIZE PROTECTION. UNPROTECTED is equivalent to TO MAXIMIZE PERFORMANCE.

TO MAXIMIZE PROTECTION This setting establishes "maximum protection mode" and offers the highest level of data protection. A transaction does not commit until all data needed to recover that transaction has been written to at least one physical standby database that is configured to use the SYNC log transport mode. If the primary database is unable to write the redo records to at least one such standby database, the primary database is shut down. This mode guarantees zero data loss, but it has the greatest potential impact on the performance and availability of the primary database.

TO MAXIMIZE AVAILABILITY This setting establishes "maximum availability mode" and offers the next highest level of data protection. A transaction does not commit until all data needed to recover that transaction has been written to at least one (physical or logical) standby database that is configured to use the SYNC log transport mode. Unlike maximum protection mode, the primary database does not shut down if it is unable to write the redo records to at least one such standby database. Instead, the protection is lowered to maximum performance mode until the fault has been corrected and the standby database has caught up with the primary database. This mode guarantees zero data loss unless the primary database fails while in maximum performance mode. Maximum availability mode provides the highest level of data protection that is possible without affecting the availability of the primary database.

TO MAXIMIZE PERFORMANCE This setting establishes "maximum performance mode" and is the default setting. A transaction commits before the data needed to recover that transaction has been written to a standby database. Therefore, some transactions may be lost if the primary database fails and you are unable to recover the redo records from the primary database. This mode provides the highest level of data protection that is possible without affecting the performance of the primary database.

To determine the current mode of the database, query the PROTECTION_MODE column of the V$DATABASE dynamic performance view.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for full information on using these standby database settings

REGISTER LOGFILE Clause

Specify the REGISTER LOGFILE clause from the standby database to manually register log files from the failed primary.

For a logical standby database, you can use this command to seed the initial starting point for a new logical standby database. Then when you issue an ALTER DATABASE START LOGICAL STANDBY APPLY INITIAL command, Oracle will use the lowest registered logfile as its starting point.

OR REPLACE Specify OR REPLACE to allow an existing archivelog entry in the standby database to be updated, for example, when its location or file specification changes. The SCNs of the entries must match exactly, and the original entry must have been created by the managed standby log transmittal mechanism.

COMMIT TO SWITCHOVER Clause

Use this clause to perform a "graceful switchover", in which the current primary database take on standby status, and one standby database becomes the primary database. In a Real Application Clusters environment, all instances other than the instance from which you issue this statement must be shutdown normally.

- Specify PHYSICAL to prepare the primary database to run in the role of a physical standby database.
- Specify LOGICAL to prepare the primary database to run in the role of a logical standby database. If you specify LOGICAL, you must then issue an ALTER DATABASE START LOGICAL STANDBY APPLY statement.

- On the primary database, specify COMMIT TO SWITCHOVER TO STANDBY to perform a graceful database switchover of the primary database to standby database status. The primary database must be open.
- On one of the standby databases, issue a COMMIT TO SWITCHOVER TO PRIMARY statement to perform a graceful switchover of this standby database to primary status. The standby database must be mounted or open in READ ONLY mode.

WITH | WITHOUT SESSION SHUTDOWN If you specify WITH SESSION SHUTDOWN, Oracle shuts down any open application sessions and rolls back uncommitted transactions as part of the execution of this statement. If you omit this clause or specify WITHOUT SESSION SHUTDOWN (which is the default), the statement will fail if any application sessions are open.

Restriction on WITH SESSION SHUTDOWN: This clause is not necessary or supported for a logical database.

WAIT | NOWAIT Specify WAIT if you want Oracle to return control after the completion of the SWITCHOVER command. Specify NOWAIT if you want Oracle to return control before the switchover operation is complete. the default is WAIT.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for full information on graceful switchover between primary and standby databases

START LOGICAL STANDBY APPLY Clause

Specify the START LOGICAL STANDBY APPLY clause to begin applying redo logs to a logical standby database.

- Specify INITIAL the first time you apply the logs to the standby database.
- Specify NEW PRIMARY after the ALTER DATABASE COMMIT TO SWITCHOVER TO LOGICAL STANDBY statement or when a standby database has completed processing logs from one primary and now a new database becomes the primary.

STOP | ABORT LOGICAL STANDBY Clause

Use this clause to stop the log apply services. This clause applies only to logical standby databases, not to physical standby databases. Use the STOP clause to stop the apply in an orderly fashion.

default_settings_clauses

Use these clauses to modify the default settings of the database.

CHARACTER SET, NATIONAL CHARACTER SET

CHARACTER SET changes the character set the database uses to store data. NATIONAL CHARACTER SET changes the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. Specify *character_set* without quotation marks. The database must be open.

Cautions:

- You cannot roll back an ALTER DATABASE CHARACTER SET or ALTER DATABASE NATIONAL CHARACTER SET statement. Therefore, you should perform a full backup before issuing either of these statements.
- Oracle Corporation recommends that you use the Character Set Scanner (CSSCAN) to analyze your data before migrating your existing database character set to a new database character set. Doing so will help you avoid losing non-ASCII data that you might not have been aware was in your database. Please see *Oracle9i Database Globalization Support Guide* for more information about CSSCAN.

Notes on Changing Character Sets:

In Oracle9*i*, CLOB data is stored as UCS-2 (two-byte fixed-width Unicode) for multibyte database character sets. For single-byte database character sets, CLOB data is stored in the database character set. When you change the database or national character set with an ALTER DATABASE statement, no data conversion is performed. Therefore, if you change the database character set from single byte to multibyte using this statement, CLOB columns will remain in the original database character set. This may introduce data inconsistency in your CLOB columns. Likewise, if you change the national character set from one Unicode set to another, your SQL NCHAR columns (NCHAR, NVARCHAR2, NCLOB) may be corrupted.

The recommended procedure for changing database character sets is:

1. Export the CLOB and SQL NCHAR datatype columns.
2. Drop the tables containing the CLOB and SQL NCHAR columns.

3. Use ALTER DATABASE statements to change the character set and national character set.
4. Reimport the CLOB and SQL NCHAR columns.

Restrictions on changing character sets:

- You must have SYSDBA system privilege, and you must start up the database in restricted mode (for example, with the SQL*Plus STARTUP RESTRICT command).
- The current character set must be a strict subset of the character set to which you change. That is, each character represented by a codepoint value in the source character set must be represented by the same codepoint value in the target character set.

> **See Also:** *Oracle9i Database Globalization Support Guide* for information on database character set migration and "Changing a Character Set: Example" on page 9-55

set_time_zone_clause

Use the SET TIME_ZONE clause to set the time zone of the database. You can specify the time zone in two ways:

- By specifying a displacement from UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The valid range of *hh:mm* is -12:00 to +14:00.
- By specifying a time zone region. To see a listing of valid region names, query the TZNAME column of the V$TIMEZONE_NAMES dynamic performance view.

> **See Also:** *Oracle9i Database Reference* for information on the dynamic performance views

Oracle normalizes all new TIMESTAMP WITH LOCAL TIME ZONE data to the time zone of the database when the data is stored on disk. Oracle does not automatically update existing data in the database to the new time zone.

After setting or changing the time zone with this clause, you must restart the database for the new time zone to take effect.

DEFAULT TEMPORARY TABLESPACE Clause

Specify this clause to change the default temporary tablespace of the database. After this operation completes, Oracle automatically reassigns to the new default

temporary tablespace all users who had been assigned to the old default temporary tablespace. You can then drop the old default temporary tablespace if you wish.

To learn the name of the current default temporary tablespace, query the PROPERTY_VALUE column of the DATABASE_PROPERTIES data dictionary table where the PROPERTY_NAME = 'DEFAULT_TEMP_TABLESPACE'.

Restrictions on default temporary tablespaces:

- The tablespace you assign or reassign as the default temporary tablespace must have a standard block size.
- If the SYSTEM tablespace is locally managed, the tablespace you specify as the default temporary tablespace must also be locally managed.

> **See Also:** "Defining a Default Temporary Tablespace: Example" on page 9-54

conversion_clauses

RESET COMPATIBILITY Clause

Specify RESET COMPATIBILITY to mark the database to be reset to an earlier version of Oracle when the database is next restarted. The database must be open.

> **Note:** RESET COMPATIBILITY works only if you have successfully disabled Oracle features that affect backward compatibility.

> **See Also:** *Oracle9i Database Migration* for more information on downgrading to an earlier version of Oracle

CONVERT Clause

Use the CONVERT clause to complete the conversion of the Oracle7 data dictionary. After you use this clause, the Oracle7 data dictionary no longer exists in the Oracle database.

> **Note:** Use this clause only when you are migrating to Oracle9*i*, and do not use this clause when the database is mounted.

See Also: *Oracle9i Database Migration*

redo_thread_clauses

Use these clauses to enable and disable the thread of redo log file groups.

ENABLE THREAD Clause

In an Oracle Real Application Clusters environment, specify ENABLE THREAD to enable the specified thread of redo log file groups. The thread must have at least two redo log file groups before you can enable it. The database must be open.

PUBLIC Specify PUBLIC to make the enabled thread available to any instance that does not explicitly request a specific thread with the initialization parameter THREAD. If you omit PUBLIC, the thread is available only to the instance that explicitly requests it with the initialization parameter THREAD.

See Also: *Oracle9i Real Application Clusters Administration* for more information on enabling and disabling threads

DISABLE THREAD Clause

Specify DISABLE THREAD to disable the specified thread, making it unavailable to all instances. The database must be open, but you cannot disable a thread if an instance using it has the database mounted.

See Also: *Oracle9i Real Application Clusters Administration* for more information on enabling and disabling threads and "Disabling and Enabling a Real Application Clusters Thread: Examples" on page 9-55

RENAME GLOBAL_NAME Clause

Specify RENAME GLOBAL_NAME to change the global name of the database. The *database* is the new database name and can be as long as eight bytes. The optional *domain* specifies where the database is effectively located in the network hierarchy. Do not use this clause when the database is mounted.

Note: Renaming your database does not change global references to your database from existing database links, synonyms, and stored procedures and functions on remote databases. Changing such references is the responsibility of the administrator of the remote databases.

> **See Also:** *Oracle9i Heterogeneous Connectivity Administrator's Guide* for more information on global names and "Changing the Global Database Name: Example" on page 9-55

security_clause

Use the *security_clause* (GUARD) to protect data in the database from being changed.

ALL Specify ALL to prevent all users other than SYS from making changes to any data in the database.

STANDBY Specify STANDBY to prevent all users other than SYS from making changes to any database object being maintained by logical standby. This setting is useful if you want report operations to be able to modify data as long as it is not being replicated by logical standby.

> **See Also:** *Oracle9i Data Guard Concepts and Administration* for information on logical standby

NONE Specify NONE if you want normal security for all data in the database.

Examples

READ ONLY / READ WRITE: Example The first statement that follows opens the database in read-only mode. The second statement returns the database to read / write mode and clears the online redo logs:

```
ALTER DATABASE OPEN READ ONLY;

ALTER DATABASE OPEN READ WRITE RESETLOGS;
```

Using Parallel Recovery Processes: Example The following statement performs tablespace recovery using parallel recovery processes:

```
ALTER DATABASE
   RECOVER TABLESPACE tbs_03
   PARALLEL;
```

Adding Redo Log File Groups: Examples The following statement adds a redo log file group with two members and identifies it with a GROUP parameter value of 3:

```
ALTER DATABASE
  ADD LOGFILE GROUP 3
    ('diska:log3.log' ,
     'diskb:log3.log') SIZE 50K;
```

The following statement adds a redo log file group containing two members to thread 5 (in a Real Application Clusters environment) and assigns it a GROUP parameter value of 4:

```
ALTER DATABASE
    ADD LOGFILE THREAD 5 GROUP 4
        ('diska:log4.log',
         'diskb:log4:log');
```

Dropping Log File Members: Example The following statement drops one redo log file member added in the previous example:

```
ALTER DATABASE
    DROP LOGFILE MEMBER 'diskb:log3.log';
```

The following statement drops all members of the redo log file group 3:

```
ALTER DATABASE DROP LOGFILE GROUP 3;
```

Adding Redo Log File Group Members: Example The following statement adds a member to the redo log file group added in the previous example:

```
ALTER DATABASE
   ADD LOGFILE MEMBER 'diskc:log3.log'
   TO GROUP 3;
```

Renaming a Log File Member: Example The following statement renames a redo log file member:

```
ALTER DATABASE
    RENAME FILE 'diskc:log3.log' TO 'diskb:log3.log';
```

The preceding statement only changes the member of the redo log group from one file to another. The statement does not actually change the name of the file `'diskc:log3.log'` to `'diskb:log3.log'`. You must perform this operation through your operating system.

Defining a Default Temporary Tablespace: Example The following statement makes the `tbs_5` tablespace the default temporary tablespace of the database. This

statement either establishes a default temporary tablespace if none was specified at create time, or replaces an existing default temporary tablespace with `temp`:

```
ALTER DATABASE
   DEFAULT TEMPORARY TABLESPACE tbs_5;
```

Disabling and Enabling a Real Application Clusters Thread: Examples The following statement disables thread 5 in a Real Application Clusters environment:

```
ALTER DATABASE
    DISABLE THREAD 5;
```

The following statement enables thread 5 in a Real Application Clusters environment, making it available to any Oracle instance that does not explicitly request a specific thread:

```
ALTER DATABASE
    ENABLE PUBLIC THREAD 5;
```

Creating a New Datafile: Example The following statement creates a new datafile '`tabspace_file04.dbf`' based on the file '`tabspace_file03.dbf`'. Before creating the new datafile, you must take the existing datafile (or the tablespace in which it resides) offline.

```
ALTER DATABASE
    CREATE DATAFILE 'tbs_f03.dbf'
                 AS 'tbs_f04.dbf';
```

Changing the Global Database Name: Example The following statement changes the global name of the database and includes both the database name and domain:

```
ALTER DATABASE
    RENAME GLOBAL_NAME TO demo.world.oracle.com;
```

Changing a Character Set: Example The following statements change the database character set and national character set to the UTF8 character set:

```
ALTER DATABASE CHARACTER SET UTF8;
ALTER DATABASE NATIONAL CHARACTER SET UTF8;
```

The database name is optional, and the character set name is specified without quotation marks.

Resizing a Datafile: Example The following statement attempts to change the size of datafile `'disk1:db1.dat'`:

```
ALTER DATABASE
    DATAFILE 'disk1:db1.dat' RESIZE 10 M;
```

Clearing a Log File: Example The following statement clears a log file:

```
ALTER DATABASE
    CLEAR LOGFILE 'diskc:log3.log';
```

Database Recovery: Examples The following statement performs complete recovery of the entire database, letting Oracle generate the name of the next archived redo log file needed:

```
ALTER DATABASE
  RECOVER AUTOMATIC DATABASE;
```

The following statement explicitly names a redo log file for Oracle to apply:

```
ALTER DATABASE
    RECOVER LOGFILE 'diskc:log3.log';
```

The following statement recovers the standby datafile `/finance/stbs_21.f`, using the corresponding datafile in the original standby database, plus all relevant archived logs and the current standby database control file:

```
ALTER DATABASE
   RECOVER STANDBY DATAFILE '/finance/stbs_21.f'
   UNTIL CONTROLFILE;
```

The following statement performs time-based recovery of the database:

```
ALTER DATABASE
    RECOVER AUTOMATIC UNTIL TIME '2001-10-27:14:00:00';
```

Oracle recovers the database until 2:00 p.m. on October 27, 2001.

For an example of recovering a tablespace, see "Using Parallel Recovery Processes: Example" on page 9-53.

Recovering a Managed Standby Database: Examples The following statement recovers the standby database in managed standby recovery mode:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE;
```

The following statement puts the database in managed standby recovery mode. The managed recovery process will wait up to 60 minutes for the next archive log:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE TIMEOUT 60;
```

If each subsequent log arrives within 60 minutes of the last log, recovery continues indefinitely or until manually terminated.

The following statement terminates the managed recovery operation:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE CANCEL IMMEDIATE;
```

The managed recovery operation terminates before the next group of redo is read from the current redo log file. Media recovery ends in the "middle" of applying redo from the current redo log file.

ALTER DIMENSION

Purpose

Use the ALTER DIMENSION statement to change the hierarchical relationships or dimension attributes of a dimension.

> **See Also:** CREATE DIMENSION on page 13-41 for more information on dimensions

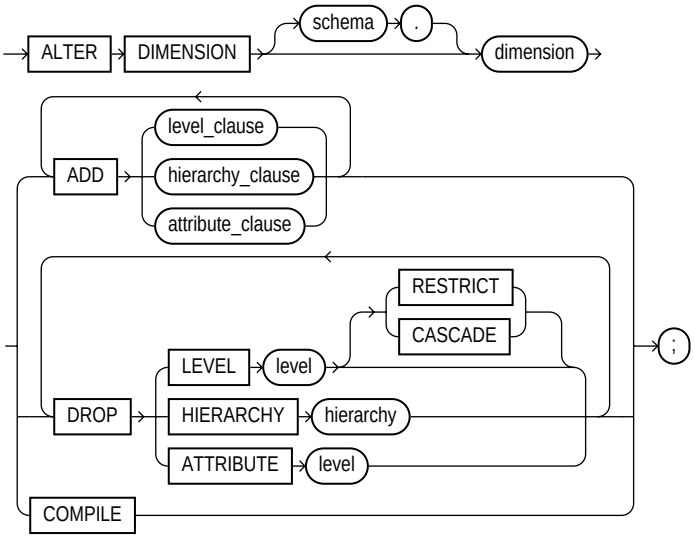
Prerequisites

The dimension must be in your schema or you must have the ALTER ANY DIMENSION system privilege to use this statement.

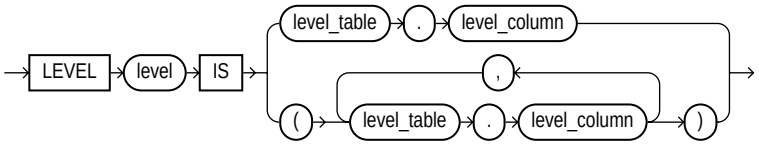
A dimension is always altered under the rights of the owner.

Syntax

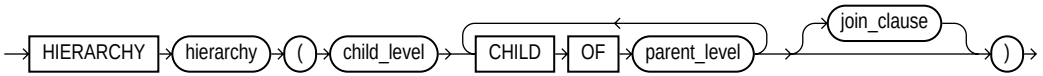
alter_dimension::=



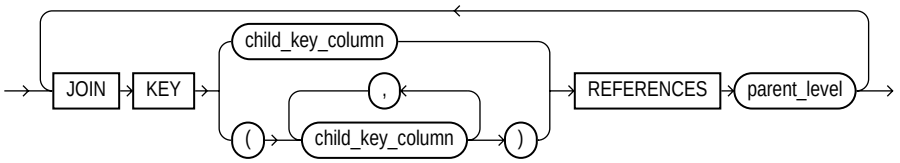
level_clause::=



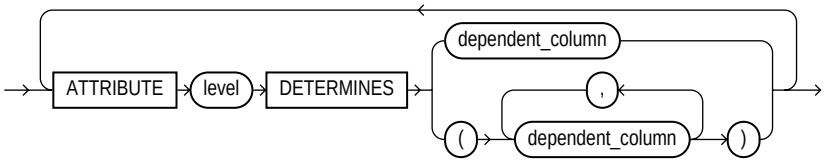
hierarchy_clause::=



join_clause::=



attribute_clause::=



Keywords and Parameters

The following keywords and parameters have meaning unique to ALTER DIMENSION. The remaining keywords and parameters have the same functionality that they have in the CREATE DIMENSION statement.

See Also: CREATE DIMENSION on page 13-41

schema

Specify the schema of the dimension you want to modify. If you do not specify *schema*, Oracle assumes the dimension is in your own schema.

dimension

Specify the name of the dimension. This dimension must already exist.

ADD

The ADD clauses let you add a level, hierarchy, or attribute to the dimension. Adding one of these elements does not invalidate any existing materialized view.

Oracle processes ADD LEVEL clauses prior to any other ADD clauses.

DROP

The DROP clauses let you drop a level, hierarchy, or attribute from the dimension. Any level, hierarchy, or attribute you specify must already exist.

Restriction on DROP: If any attributes or hierarchies reference a level, you cannot drop the level until you either drop all the referencing attributes and hierarchies or specify CASCADE.

CASCADE Specify CASCADE if you want Oracle to drop any attributes or hierarchies that reference the level, along with the level itself.

RESTRICT Specify RESTRICT if you want to prevent Oracle from dropping a level that is referenced by any attributes or hierarchies. This is the default.

COMPILE

Specify COMPILE to explicitly recompile an invalidated dimension. Oracle automatically compiles a dimension when you issue an ADD clause or DROP clause. However, if you alter an object referenced by the dimension (for example, if you drop and then re-create a table referenced in the dimension), the dimension will be invalidated, and you must recompile it explicitly.

Example

Modifying a Dimension: Examples The following examples modify the customers_dim dimension in the sample schema sh:

```
ALTER DIMENSION customers_dim
   DROP ATTRIBUTE country;

ALTER DIMENSION customers_dim
   ADD LEVEL zone IS customers.cust_postal_code
   ADD ATTRIBUTE zone DETERMINES (cust_city);
```

ALTER FUNCTION

Purpose

Use the ALTER FUNCTION statement to recompile an invalid standalone stored function. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The ALTER FUNCTION statement is similar to ALTER PROCEDURE on page 9-126. For information on how Oracle recompiles functions and procedures, see *Oracle9i Database Concepts*.

Note: This statement does not change the declaration or definition of an existing function. To redeclare or redefine a function, use the CREATE FUNCTION statement with the OR REPLACE clause; see CREATE FUNCTION on page 13-49.

Prerequisites

The function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

alter_function::=

Keywords and Parameters

schema

Specify the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.

function

Specify the name of the function to be recompiled.

COMPILE

Specify COMPILE to cause Oracle to recompile the function. The COMPILE keyword is required. If Oracle does not compile the function successfully, you can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

During recompilation, Oracle drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the REUSE SETTINGS clause.

DEBUG

Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

REUSE SETTINGS

Specify REUSE SETTINGS to prevent Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation.

If you specify both DEBUG and REUSE SETTINGS, Oracle sets the persistently stored value of the PLSQL_COMPILER_FLAGS parameter to INTERPRETED, DEBUG. No other compiler switch values are changed.

> **See Also:** *PL/SQL User's Guide and Reference* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on the interaction of the PLSQL_COMPILER_FLAGS parameter with the COMPILE clause

Example

Recompiling a Function: Example To explicitly recompile the function get_bal owned by the sample user oe, issue the following statement:

```
ALTER FUNCTION oe.get_bal
   COMPILE;
```

If Oracle encounters no compilation errors while recompiling get_bal, get_bal becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling get_bal results in compilation errors, Oracle returns an error, and get_bal remains invalid.

Oracle also invalidates all objects that depend upon `get_bal`. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

ALTER INDEX

Purpose

Use the ALTER INDEX statement to change or rebuild an existing index.

> **See Also:** CREATE INDEX on page 13-62 for information on creating an index

Prerequisites

The index must be in your own schema or you must have ALTER ANY INDEX system privilege.

To execute the MONITORING USAGE clause, the index must be in your own schema.

To modify a domain index, you must have EXECUTE object privilege on the indextype of the index.

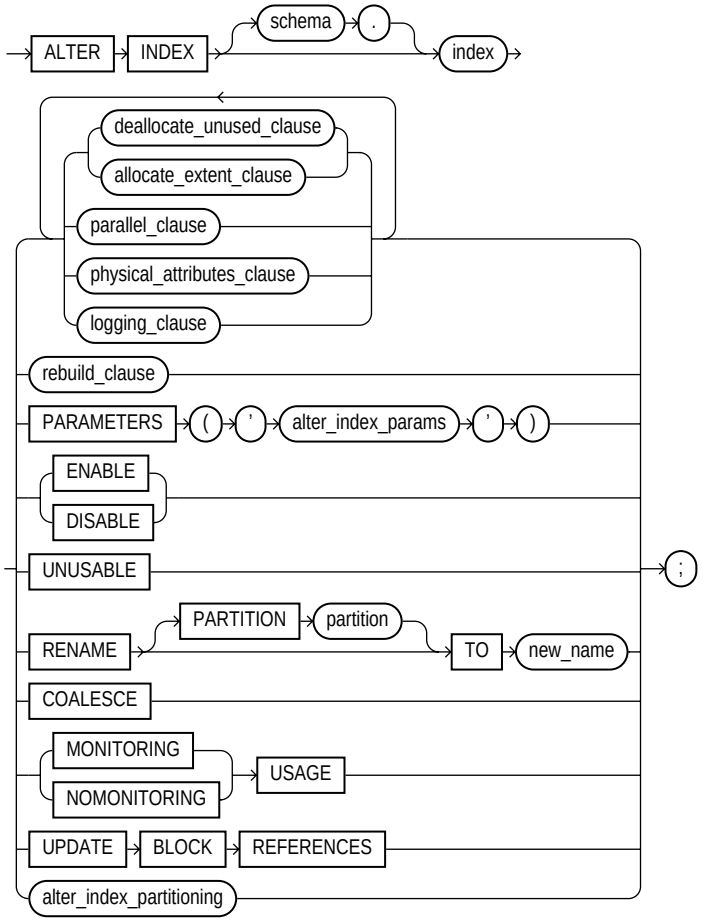
Schema object privileges are granted on the parent index, not on individual index partitions or subpartitions.

You must have tablespace quota to modify, rebuild, or split an index partition or to modify or rebuild an index subpartition.

> **See Also:** CREATE INDEX on page 13-62 and *Oracle9i Data Cartridge Developer's Guide* for information on domain indexes

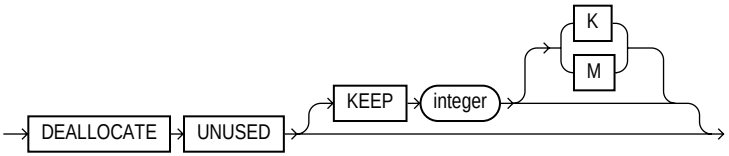
Syntax

alter_index::=

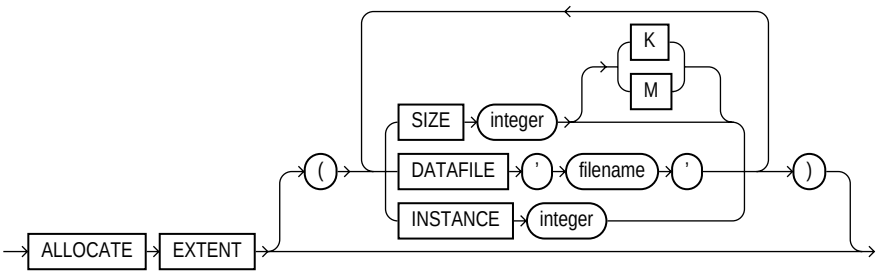


(*deallocate_unused_clause::=* on page 9-66, *allocate_extent_clause::=* on page 9-66, *parallel_clause::=* on page 9-66, *physical_attributes_clause::=* on page 9-66, *logging_clause::=* on page 7-46, *rebuild_clause::=* on page 9-67, *alter_index_partitioning::=* on page 9-68)

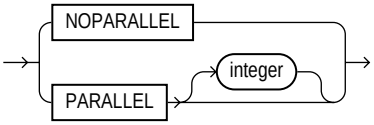
deallocate_unused_clause::=



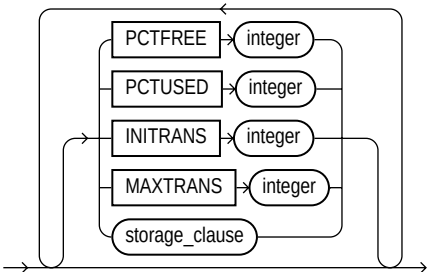
allocate_extent_clause::=



parallel_clause::=

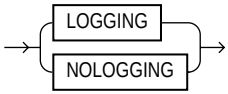


physical_attributes_clause::=

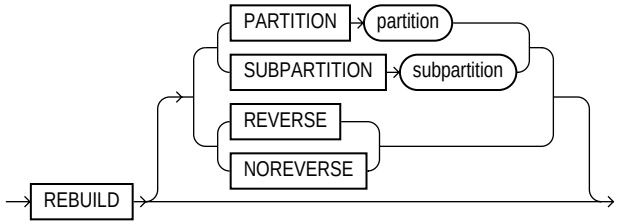


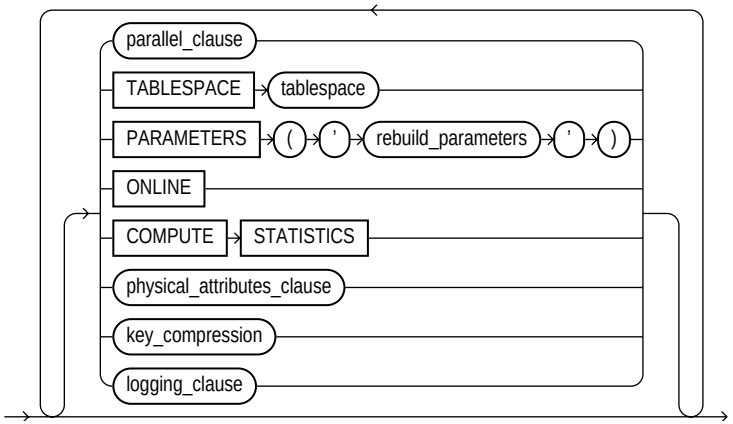
(*storage_clause::=* on page 7-58)

logging_clause::=



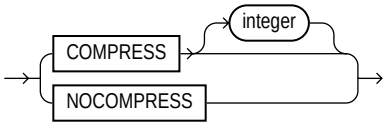
rebuild_clause::=



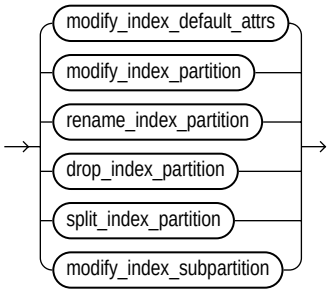


(*physical_attributes_clause::=* on page 9-66, *key_compression::=* on page 9-67, *logging_clause::=* on page 7-46)

key_compression::=

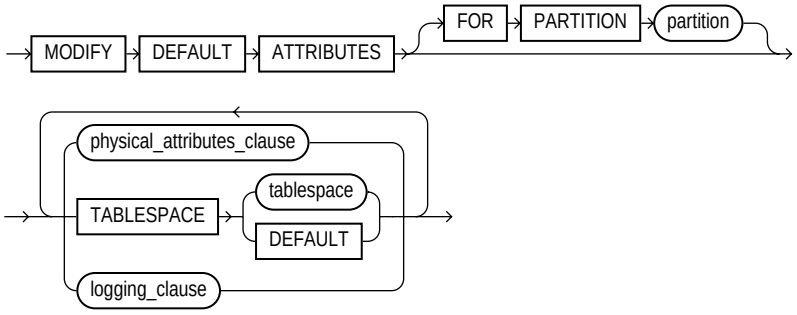


alter_index_partitioning::=



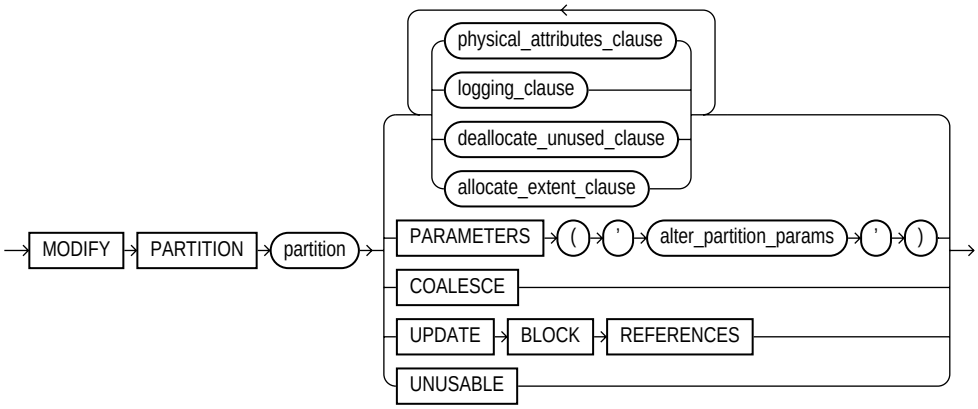
(*modify_index_default_attrs::=* on page 9-68, *modify_index_partition::=* on page 9-69, *rename_index_partition::=* on page 9-69, *drop_index_partition::=* on page 9-69, *split_index_partition::=* on page 9-69, *modify_index_subpartition::=* on page 9-70)

modify_index_default_attrs::=



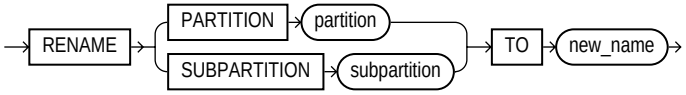
(*physical_attributes_clause::=* on page 9-66, *logging_clause::=* on page 7-46)

modify_index_partition::=

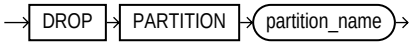


(*physical_attributes_clause::=* on page 9-66, *logging_clause::=* on page 7-46, *allocate_extent_clause::=* on page 9-66, *deallocate_unused_clause::=* on page 9-66)

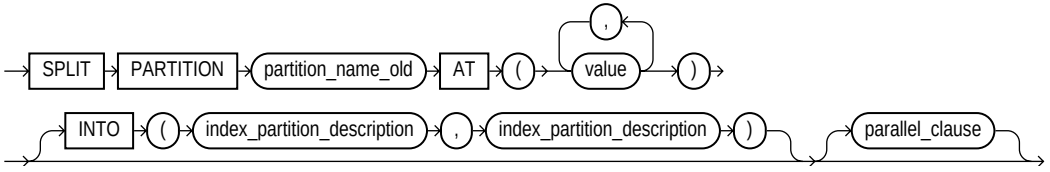
rename_index_partition::=



drop_index_partition::=

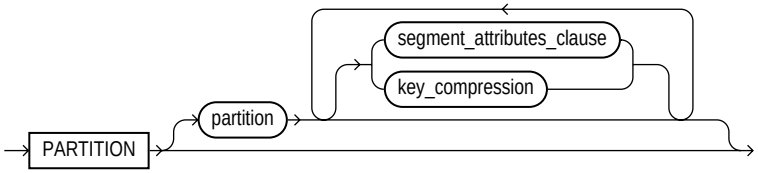


split_index_partition::=



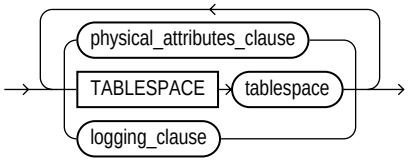
(*parallel_clause::=* on page 9-66)

index_partition_description::=



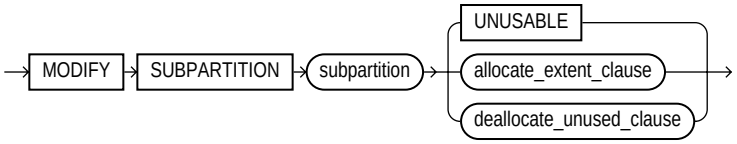
(*key_compression::=* on page 9-67)

segment_attributes_clause::=



(*physical_attributes_clause::=* on page 9-66, *logging_clause::=* on page 7-46)

modify_index_subpartition::=



(*allocate_extent_clause::=* on page 9-66, *deallocate_unused_clause::=* on page 9-66)

Keywords and Parameters

schema

Specify the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index

Specify the name of the index to be altered.

Restrictions on indexes:

- If *index* is a domain index, you can specify only the PARAMETERS clause, the RENAME clause, the *rebuild_clause* (with or without the PARAMETERS clause), the *parallel_clause*, or the UNUSABLE clause. No other clauses are valid.
- You cannot alter or rename a domain index that is marked LOADING or FAILED. If an index is marked FAILED, the only clause you can specify is REBUILD.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for information on the LOADING and FAILED states of domain indexes

deallocate_unused_clause

Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of the index and make the freed space available for other segments in the tablespace.

If *index* is range-partitioned or hash-partitioned, Oracle deallocates unused space from each index partition. If *index* is a local index on a composite-partitioned table, Oracle deallocates unused space from each index subpartition.

Restrictions on deallocating space:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify this clause and also specify the *rebuild_clause*.

> **See Also:** *deallocate_unused_clause* on page 7-37 for a full description of this clause

KEEP *integer* The KEEP clause lets you specify the number of bytes above the high water mark that the index will have after deallocation. If the number of remaining extents are less than MINEXTENTS, then MINEXTENTS is set to the current number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is set to the value of the current initial extent. If you omit KEEP, all unused space is freed.

> **See Also:** ALTER TABLE on page 11-2 for a complete description of this clause

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocate a new extent for the index. For a local index on a hash-partitioned table, Oracle allocates a new extent for each partition of the index.

Restriction on allocating extents: You cannot specify this clause for an index on a temporary table or for a range-partitioned or composite-partitioned index.

> **See Also:** *allocate_extent_clause* on page 7-2 for a full description of this clause

parallel_clause

Use the PARALLEL clause to change the default degree of parallelism for queries and DML on the index.

Restriction on the *parallel_clause***:** You cannot specify this clause for an index on a temporary table.

> **Note:** The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56 and "Enabling Parallel Queries: Example" on page 9-85

physical_attributes_clause

Use the *physical_attributes_clause* to change the values of parameters for a nonpartitioned index, all partitions and subpartitions of a partitioned index, a specified partition, or all subpartitions of a specified partition.

See Also:

- the physical attributes parameters in CREATE TABLE on page 15-7
- "Modifying Real Attributes: Example" on page 9-84 and "Changing MAXEXTENTS: Example" on page 9-85

Restrictions on the *physical_attributes_clause*:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify the PCTUSED parameter at all when altering an index.
- You can specify the PCTFREE parameter only as part of the *rebuild_clause*, the *modify_index_default_attrs* clause, or the *split_partition_clause*.

storage_clause

Use the *storage_clause* to change the storage parameters for a nonpartitioned index, index partition, or all partitions of a partitioned index, or default values of these parameters for a partitioned index.

See Also: *storage_clause* on page 7-56

logging_clause

Use the *logging_clause* to specify whether subsequent Direct Loader (SQL*Loader) and direct-path INSERT operations against a nonpartitioned index, a range or hash index partition, or all partitions or subpartitions of a composite-partitioned index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

An index segment can have logging attributes different from those of the base table and different from those of other index segments for the same base table.

Restriction on the *logging_clause*: You cannot specify this clause for an index on a temporary table.

See Also:

- *logging_clause* on page 7-45 for a full description of this clause
- *Oracle9i Database Concepts* and the *Oracle9i Data Warehousing Guide* for more information about LOGGING and parallel DML

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with LOGGING and NOLOGGING, respectively. Although RECOVERABLE and UNRECOVERABLE are supported for backward compatibility, Oracle Corporation strongly recommends that you use the LOGGING and NOLOGGING keywords.

RECOVERABLE is not a valid keyword for creating partitioned tables or LOB storage characteristics. UNRECOVERABLE is not a valid keyword for creating partitioned or index-organized tables. Also, it can be specified only with the AS subquery clause of CREATE INDEX.

rebuild_clause

Use the *rebuild_clause* to re-create an existing index or one of its partitions or subpartitions. If index is marked UNUSABLE, a successful rebuild will mark it USABLE. For a function-based index, this clause also enables the index. If the function on which the index is based does not exist, the rebuild statement will fail.

Restrictions on rebuilding indexes:

- You cannot rebuild an index on a temporary table.
- You cannot rebuild a bitmap index that is marked INVALID. Instead, you must drop and then re-create it.
- You cannot rebuild an entire partitioned index. You must rebuild each partition or subpartition, as described for the PARTITION clause.
- You cannot also specify the *deallocate_unused_clause* in this statement.
- You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or for a partition (ALTER INDEX ... MODIFY PARTITION). You can specify PCTFREE in all other forms of the ALTER INDEX statement.
- For a domain index:
  - You can specify only the PARAMETERS clause (either for the index or for a partition of the index) or the *parallel_clause*. No other rebuild clauses are valid.

  - You can rebuild the index only if index is not marked IN_PROGRESS.
  - You can rebuild the index partitions only if index is not marked IN_PROGRESS or FAILED and partition is not marked IN_PROGRESS.
- You cannot rebuild a local index, but you can rebuild a partition of a local index (ALTER INDEX ... REBUILD PARTITION).
- For a local index on a hash partition or subpartition, the only parameter you can specify is TABLESPACE.

PARTITION Clause

Use the PARTITION clause to rebuild one partition of an index. You can also use this clause to move an index partition to another tablespace or to change a create-time physical attribute.

Note: The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle9i Database Administrator's Guide* for a discussion of these restrictions.

Restriction on REBUILD PARTITION: You cannot specify this clause for a local index on a composite-partitioned table. Instead, use the REBUILD SUBPARTITION clause.

See Also: *Oracle9i Database Administrator's Guide* for more information about partition maintenance operations and "Rebuilding Unusable Index Partitions: Example" on page 9-85

SUBPARTITION Clause

Use the SUBPARTITION clause to rebuild one subpartition of an index. You can also use this clause to move an index subpartition to another tablespace. If you do not specify TABLESPACE, the subpartition is rebuilt in the same tablespace.

Note: The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle9i Database Administrator's Guide* for a discussion of these restrictions.

Restrictions on the SUBPARTITION clause:

- The only parameters you can specify for a subpartition are TABLESPACE and the *parallel_clause*.
- You cannot rebuild the subpartition of a list partition.

REVERSE | NOREVERSE

Indicate whether the bytes of the index block are stored in reverse order:

- REVERSE stores the bytes of the index block in reverse order and excludes the rowid when the index is rebuilt.
- NOREVERSE stores the bytes of the index block without reversing the order when the index is rebuilt. Rebuilding a REVERSE index without the NOREVERSE keyword produces a rebuilt, reverse-keyed index.

Restrictions on REVERSE:

- You cannot reverse a bitmap index or an index-organized table.
- You cannot specify REVERSE or NOREVERSE for a partition or subpartition.

> **See Also:** "Storing Index Blocks in Reverse Order: Example" on page 9-84

parallel_clause

Use the *parallel_clause* to parallelize the rebuilding of the index.

> **See Also:** "Rebuilding an Index in Parallel: Example" on page 9-84

TABLESPACE Clause

Specify the tablespace where the rebuilt index, index partition, or index subpartition will be stored. The default is the default tablespace where the index or partition resided before you rebuilt it.

COMPRESS | NOCOMPRESS

Specify COMPRESS to enable key compression, which eliminates repeated occurrence of key column values. Use *integer* to specify the prefix length (number of prefix columns to compress).

- For unique indexes, the range of valid prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the range of valid prefix length values is from 1 to the number of key columns. The default prefix length is number of key columns.

Oracle compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns.

Restriction on key compression: You cannot specify `COMPRESS` for a bitmap index.

Specify `NOCOMPRESS` to disable key compression. This is the default.

ONLINE Clause

Specify `ONLINE` to allow DML operations on the table or partition during rebuilding of the index.

Restrictions on the ONLINE clause:

- Parallel DML is not supported during online index building. If you specify `ONLINE` and then issue parallel DML statements, Oracle returns an error.
- You cannot specify `ONLINE` for a bitmap index or a cluster index.
- For a unique index on an index-organized table, the number of index key columns plus the number of primary key columns in the index-organized table cannot exceed 32.

COMPUTE STATISTICS Clause

Specify `COMPUTE STATISTICS` if you want to collect statistics at relatively little cost during the rebuilding of an index. These statistics are stored in the data dictionary for ongoing use by the optimizer in choosing a plan of execution for SQL statements.

The types of statistics collected depend on the type of index you are rebuilding.

Note: If you create an index using another index (instead of a table), the original index might not provide adequate statistical information. Therefore, Oracle generally uses the base table to compute the statistics, which will improve the statistics but may negatively affect performance.

Additional methods of collecting statistics are available in PL/SQL packages and procedures

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* and "Collecting Index Statistics: Example" on page 9-84

logging_clause

Specify whether the ALTER INDEX ... REBUILD operation will be logged.

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

PARAMETERS Clause

The PARAMETERS clause applies only to domain indexes. This clause specifies the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The maximum length of the parameter string is 1000 characters.

If you are altering or rebuilding an entire index, the string must refer to index-level parameters. If you are rebuilding a partition of the index, the string must refer to partition-level parameters.

If index is marked UNUSABLE, modifying the parameters alone does not make it USABLE. You must also rebuild the UNUSABLE index to make it usable.

Note: If you have installed Oracle Text, you can rebuild your Oracle Text domain indexes using parameters specific to that product. For more information on those parameters, please refer to *Oracle Text Reference*.

Restrictions on the PARAMETERS clause:

- You can specify this clause only for a domain index.
- You can modify index partitions only if index is not marked IN_PROGRESS or FAILED, no index partitions are marked IN_PROGRESS, and the partition being modified is not marked FAILED.

See Also:

- *Oracle9i Data Cartridge Developer's Guide* for more information on indextype routines.
- CREATE INDEX on page 13-62 for more information on domain indexes

ENABLE Clause

ENABLE applies only to a function-based index that has been disabled because a user-defined function used by the index was dropped or replaced. This clause enables such an index if these conditions are true:

- The function is currently valid
- The signature of the current function matches the signature of the function when the index was created
- The function is currently marked as DETERMINISTIC

Restriction on the ENABLE clause: You cannot specify any other clauses of ALTER INDEX in the same statement with ENABLE.

DISABLE Clause

DISABLE applies only to a function-based index. This clause enables you to disable the use of a function-based index. You might want to do so, for example, while working on the body of the function. Afterward you can either rebuild the index or specify another ALTER INDEX statement with the ENABLE keyword.

UNUSABLE Clause

Specify UNUSABLE to mark the index or index partition(s) or index subpartition(s) UNUSABLE. An unusable index must be rebuilt, or dropped and re-created, before it can be used. While one partition is marked UNUSABLE, the other partitions of the index are still valid. You can execute statements that require the index if the statements do not access the unusable partition. You can also split or rename the unusable partition before rebuilding it.

Restriction on the UNUSABLE clause: You cannot specify this clause for an index on a temporary table.

RENAME Clause

Specify RENAME TO to rename an index or a partition of an index. The *new_index_name* is a single identifier and does not include the schema name.

Restrictions on the RENAME clause:

- For a domain index, neither *index* nor any partitions of *index* can be marked IN_PROGRESS or FAILED.
- For a partition of a domain index, *index* must not be marked IN_PROGRESS or FAILED, none of the partitions can be marked IN_PROGRESS, and the partition you are renaming must not be marked FAILED.

> **See Also:** "Renaming an Index: Example" on page 9-85 and "Renaming an Index Partition: Example" on page 9-85

COALESCE Clause

Specify COALESCE to instruct Oracle to merge the contents of index blocks where possible to free blocks for reuse.

Restrictions on the COALESCE clause:

- You cannot specify this clause for an index on a temporary table.
- Do not specify this clause for the primary key index of an index-organized table. Instead use the COALESCE clause of ALTER TABLE.

> **See Also:**
>
> - *Oracle9i Database Administrator's Guide* for more information on space management and coalescing indexes
> - COALESCE on page 11-108 for information on coalescing space of an index-organized table

MONITORING USAGE | NOMONITORING USAGE

Use this clause to determine whether Oracle should monitor index use.

- Specify MONITORING USAGE to begin monitoring the index. Oracle first clears existing information on index usage, and then monitors the index for use until a subsequent ALTER INDEX ... NOMONITORING USAGE statement is executed.
- To terminate monitoring of the index, specify NOMONITORING USAGE.

To see whether the index has been used since this ALTER INDEX ... NOMONITORING USAGE statement was issued, query the USED column of the V$OBJECT_USAGE dynamic performance view.

> **See Also:** *Oracle9i Database Reference* for information on the data dictionary and dynamic performance views

UPDATE BLOCK REFERENCES Clause

The UPDATE BLOCK REFERENCES clause is valid only for normal and domain indexes on index-organized tables. Specify this clause to update all the stale "guess" data block addresses stored as part of the index row with the correct database address for the corresponding block identified by the primary key.

Note: For a domain index, Oracle executes the `ODCIIndexAlter` routine with the `alter_option` parameter set to `AlterIndexUpdBlockRefs`. This routine enables the cartridge code to update the stale "guess" data block addresses in the index.

Restriction on the UPDATE BLOCK REFERENCES clause: You cannot combine this clause with any other clause of ALTER INDEX.

alter_index_partitioning

The partitioning clauses of the ALTER INDEX statement are valid only for partitioned indexes.

Note: The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle9i Database Administrator's Guide* for a discussion of these restrictions.

Restrictions on altering index partitions:

- You cannot specify any of these clauses for an index on a temporary table.
- You can combine several operations on the base index into one ALTER INDEX statement (except RENAME and REBUILD), but you cannot combine partition operations with other partition operations or with operations on the base index.

modify_index_default_attrs

Specify new values for the default attributes of a partitioned index.

Restriction on the *modify_index_default_attrs* **clause:** The only attribute you can specify for an index on a hash-partitioned or composite-partitioned table is TABLESPACE.

TABLESPACE Specify the default tablespace for new partitions of an index or subpartitions of an index partition.

logging_clause Specify the default logging attribute of a partitioned index or an index partition.

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

FOR PARTITION Use the FOR PARTITION clause to specify the default attributes for the subpartitions of a partition of a local index on a composite-partitioned table.

Restriction on the FOR PARTITION clause: You cannot specify FOR PARTITION for a list partition.

> **See Also:** "Modifying Default Attributes: Example" on page 9-86

modify_index_partition

Use the *modify_index_partition* clause to modify the real physical attributes, logging attribute, or storage characteristics of index partition *partition* or its subpartitions.

UPDATE BLOCK REFERENCES The UPDATE BLOCK REFERENCES clause is valid only for normal indexes on index-organized tables. Use this clause to update all stale "guess" data block addresses stored in the secondary index partition.

Restrictions on updating block references:

- You cannot specify the *physical_attributes_clause* for an index on a hash-partitioned table.
- You cannot specify UPDATE BLOCK REFERENCES with any other clause in ALTER INDEX.

Note: If the index is a local index on a composite-partitioned table, the changes you specify here will override any attributes specified earlier for the subpartitions of index, as well as establish default values of attributes for future subpartitions of that partition. To change the default attributes of the partition without overriding the attributes of subpartitions, use ALTER TABLE ... MODIFY DEFAULT ATTRIBUTES OF PARTITION.

See Also: "Marking an Index Unusable: Examples" on page 9-85

rename_index_partition

Use the *rename_index_partition* clauses to rename index partition or subpartition to *new_name*.

Restriction on renaming index partitions: You cannot rename the subpartition of a list partition.

drop_index_partition

Use the *drop_index_partition* clause to remove a partition and the data in it from a partitioned global index. When you drop a partition of a global index, Oracle marks the index's next partition UNUSABLE. You cannot drop the highest partition of a global index.

See Also: "Dropping an Index Partition: Example" on page 9-86

split_index_partition

Use the *split_index_partition* clause to split a partition of a global partitioned index into two partitions, adding a new partition to the index.

Splitting a partition marked UNUSABLE results in two partitions, both marked UNUSABLE. You must rebuild the partitions before you can use them.

Splitting a usable partition results in two partitions populated with index data. Both new partitions are usable.

AT Clause Specify the new noninclusive upper bound for *split_partition_1*. The *value_list* must evaluate to less than the presplit partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one).

INTO Clause Specify (optionally) the name and physical attributes of each of the two partitions resulting from the split.

See Also: "Splitting a Partition Example" on page 9-86

modify_index_subpartition

Use the *modify_index_subpartition* clause to mark UNUSABLE or allocate or deallocate storage for a subpartition of a local index on a composite-partitioned

table. All other attributes of such a subpartition are inherited from partition-level default attributes.

Examples

Storing Index Blocks in Reverse Order: Example The following statement rebuilds index `ord_customer_ix` (created in "Creating an Index: Example" on page 13-83) so that the bytes of the index block are stored in reverse order:

```
ALTER INDEX ord_customer_ix REBUILD REVERSE;
```

Collecting Index Statistics: Example The following statement collects statistics on the nonpartitioned `ord_customer_ix` index:

```
ALTER INDEX ord_customer_ix REBUILD COMPUTE STATISTICS;
```

The type of statistics collected depends on the type of index you are rebuilding.

> **See Also:** *Oracle9i Database Concepts*

Rebuilding an Index in Parallel: Example The following statement causes the index to be rebuilt from the existing index by using parallel execution processes to scan the old and to build the new index:

```
ALTER INDEX ord_customer_ix REBUILD PARALLEL;
```

Modifying Real Attributes: Example The following statement alters the `oe.cust_lname_ix` index so that future data blocks within this index use 5 initial transaction entries and an incremental extent of 100 kilobytes:

```
/* Unless you change the default tablespace of sample user oe,
   or specify different tablespace storage for the index, this
   example fails because the default tablespace originally assigned
   to oe is locally managed.
*/
ALTER INDEX oe.cust_lname_ix
    INITRANS 5
    STORAGE (NEXT 100K);
```

If the `oe.cust_lname_ix` index were partitioned, this statement would also alter the default attributes of future partitions of the index. New partitions added in the future would then use 5 initial transaction entries and an incremental extent of 100K.

Enabling Parallel Queries: Example The following statement sets the parallel attributes for index upper_ix (created in "Creating a Function-Based Index: Example" on page 13-85) so that scans on the index will be parallelized:

```
ALTER INDEX upper_ix PARALLEL;
```

Renaming an Index: Example The following statement renames an index:

```
ALTER INDEX upper_ix RENAME TO upper_name_ix;
```

Marking an Index Unusable: Examples The following statements use the cost_ix index, which was created in "Creating a Global Partitioned Index: Example" on page 13-87. Partition p1 of that index was dropped in "Dropping an Index Partition: Example" on page 9-86. The first statement marks the marks index partition p2 as UNUSABLE:

```
ALTER INDEX cost_ix
   MODIFY PARTITION p2 UNUSABLE;
```

The next statement marks the entire index cost_ix as UNUSABLE:

```
ALTER INDEX cost_ix UNUSABLE;
```

Rebuilding Unusable Index Partitions: Example The following statements rebuild partitions p2 and p3 of the cost_ix index, making the index once more usable: The rebuilding of partition p3 will not be logged:

```
ALTER INDEX cost_ix
   REBUILD PARTITION p2;
ALTER INDEX cost_ix
   REBUILD PARTITION p3 NOLOGGING;
```

Changing MAXEXTENTS: Example The following statement changes the maximum number of extents for partition p3 and changes the logging attribute:

```
/* This example will fail if the tablespace in which partition p3
   resides is locally managed.
*/
ALTER INDEX cost_ix MODIFY PARTITION p3
   STORAGE(MAXEXTENTS 30) LOGGING;
```

Renaming an Index Partition: Example The following statement renames an index partition of the cost_ix index (created in "Creating a Global Partitioned Index: Example" on page 13-87):

```
ALTER INDEX cost_ix
```

```
   RENAME PARTITION p3 TO p3_Q3;
```

Splitting a Partition Example The following statement splits partition p2 of index cost_ix (created in "Creating a Global Partitioned Index: Example" on page 13-87) into p2a and p2b:

```
ALTER INDEX cost_ix
   SPLIT PARTITION p2 AT (1500)
   INTO ( PARTITION p2a TABLESPACE tbs_01 LOGGING,
          PARTITION p2b TABLESPACE tbs_02);
```

Dropping an Index Partition: Example The following statement drops index partition p1 from the cost_ix index:

```
ALTER INDEX cost_ix
   DROP PARTITION p1;
```

Modifying Default Attributes: Example The following statement alters the default attributes of local partitioned index prod_idx, which was created in "Creating an Index on a Hash-Partitioned Table: Example." on page 13-87. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100K:

```
ALTER INDEX prod_idx
      MODIFY DEFAULT ATTRIBUTES INITRANS 5 STORAGE (NEXT 100K);
```

ALTER INDEXTYPE

Purpose

Use the ALTER INDEXTYPE statement to add or drop an operator of the indextype or to modify the implementation type or change the properties of the indextype.

Prerequisites

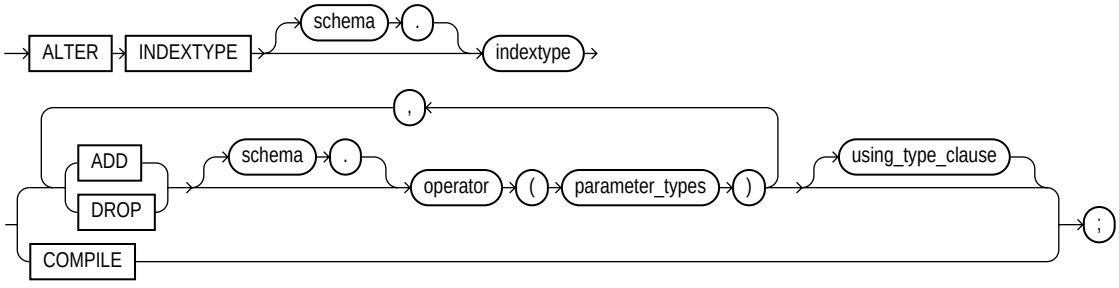
To alter an indextype in your own or another schema, you must have the ALTER ANY INDEXTYPE system privilege.

To add a new operator, you must have the EXECUTE object privilege on the operator.

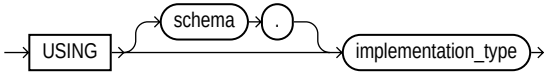
To change the implementation type, you must have the EXECUTE object privilege on the new implementation type.

Syntax

alter_indextype::=



using_type_clause::=



Keywords and Parameters

schema

Specify the name of the schema in which the indextype resides. If you omit *schema*, Oracle assumes the indextype is in your own schema.

indextype

Specify the name of the indextype to be modified.

ADD | DROP

Use the ADD or DROP clause to add or drop an operator.

- For *schema*, specify the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.
- For *operator*, specify the name of the operator supported by the indextype.

  All the operators listed in this clause should be valid operators.
- For *parameter_type*, list the types of parameters to the operator.

USING Clause

The USING clause lets you specify a new type to provide the implementation for the indextype.

COMPILE

Use this clause to recompile the indextype explicitly. This clause is required only after some upgrade operations, because Oracle normally recompiles the indextype automatically.

Examples

Altering an Indextype: Example The following example adds another operator binding to the TextIndexType indextype created in the CREATE INDEXTYPE statement. TextIndexType can now support a new operator lob_contains with the bindings(CLOB, CLOB):

```
ALTER INDEXTYPE TextIndexType ADD lob_contains(CLOB, CLOB);
```

ALTER JAVA

Purpose

Use the ALTER JAVA statement to force the resolution of a Java class schema object or compilation of a Java source schema object. (You cannot call the methods of a Java class before all its external references to Java names are associated with other classes.)

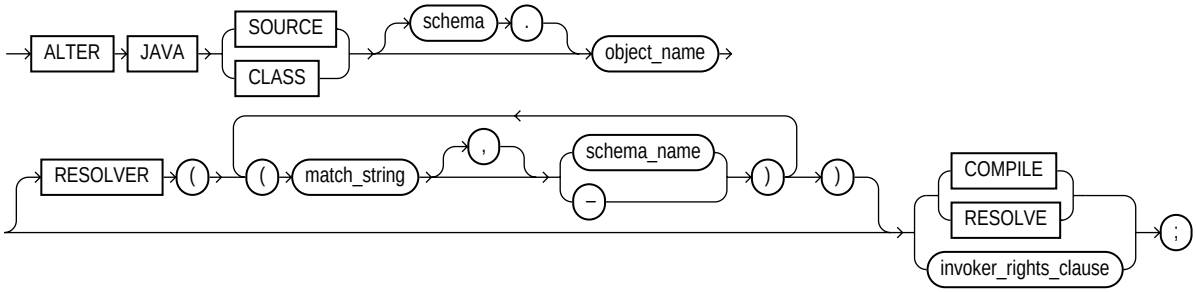
> **See Also:** *Oracle9i Java Stored Procedures Developer's Guide* for more information on resolving Java classes and compiling Java sources

Prerequisites

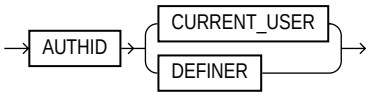
The Java source or class must be in your own schema, or you must have the ALTER ANY PROCEDURE system privilege. You must also have the EXECUTE object privilege on Java classes.

Syntax

alter_java::=



invoker_rights_clause::=



Keywords and Parameters

JAVA SOURCE

Use ALTER JAVA SOURCE to compile a Java source schema object.

JAVA CLASS

Use ALTER JAVA CLASS to resolve a Java class schema object.

object_name

Specify a previously created Java class or source schema object. Use double quotation marks to preserve lower- or mixed-case names.

RESOLVER

The RESOLVER clause lets you specify how schemas are searched for referenced fully specified Java names, using the mapping pairs specified when the Java class or source was created.

> **See Also:** CREATE JAVA on page 13-94 and "Resolving a Java Class: Example" on page 9-91

RESOLVE | COMPILE

RESOLVE and COMPILE are synonymous keywords. They let you specify that Oracle should attempt to resolve the primary Java class schema object.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the methods of the class execute with the privileges and in the schema of the user who defined it or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER Specify CURRENT_USER if you want the methods of the class to execute with the privileges of CURRENT_USER. This clause is the default and creates an "invoker-rights class."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER Specify DEFINER if you want the methods of the class to execute with the privileges of the user who defined it.

This clause also specifies that external names resolve in the schema where the methods reside.

See Also:

- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *Oracle9i Java Stored Procedures Developer's Guide*

Example

Resolving a Java Class: Example The following statement forces the resolution of a Java class:

```
ALTER JAVA CLASS "Agent"
   RESOLVER (("/home/java/bin/*" pm)(* public))
   RESOLVE;
```

ALTER MATERIALIZED VIEW

Purpose

A materialized view is a database object that contains the results of a query. The FROM clause of the query can name tables, views, and other materialized views. Collectively these are called **master tables** (a replication term) or **detail tables** (a data warehouse term). This reference uses "master tables" for consistency. The databases containing the master tables are called the **master databases**.

Use the ALTER MATERIALIZED VIEW statement to modify an existing materialized view in one or more of the following ways:

- To change its storage characteristics
- To change its refresh method, mode, or time
- To alter its structure so that it is a different type of materialized view
- To enable or disable query rewrite.

Note: The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

See Also:

- CREATE MATERIALIZED VIEW on page 14-5 for more information on creating materialized views
- *Oracle9i Replication* for information on materialized views in a replication environment
- *Oracle9i Data Warehousing Guide* for information on materialized views in a data warehousing environment

Prerequisites

The privileges required to alter a materialized view should be granted directly, as follows:

The materialized view must be in your own schema, or you must have the ALTER ANY MATERIALIZED VIEW system privilege.

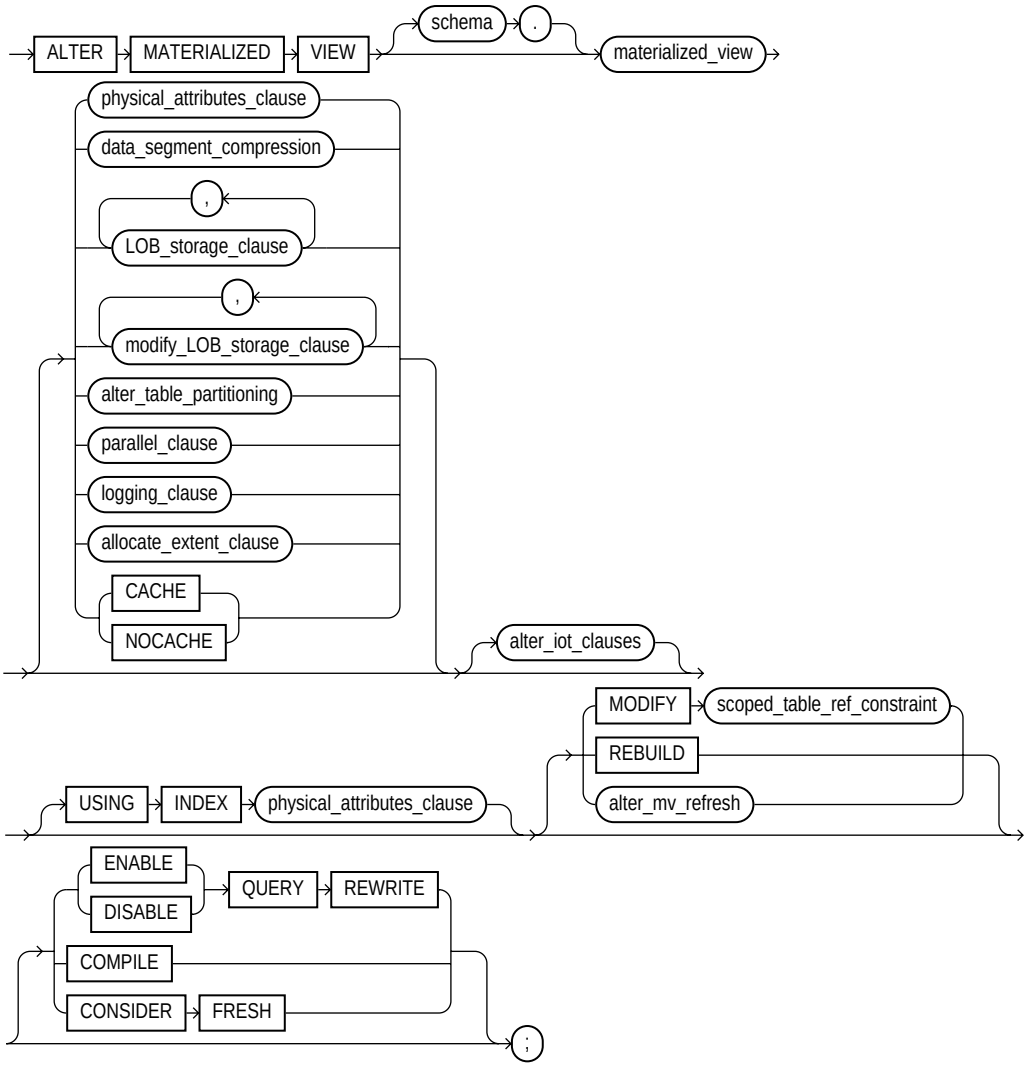
To enable a materialized view for query rewrite:

- If all of the master tables in the materialized view are in your schema, you must have the QUERY REWRITE privilege.
- If any of the master tables are in another schema, you must have the GLOBAL QUERY REWRITE privilege.
- If the materialized view is in another user's schema, both you and the owner of that schema must have the appropriate QUERY REWRITE privilege, as described in the preceding two items. In addition, the owner of the materialized view must have SELECT access to any master tables that the materialized view owner does not own.

See Also: *Oracle9i Replication* and *Oracle9i Data Warehousing Guide*

Syntax

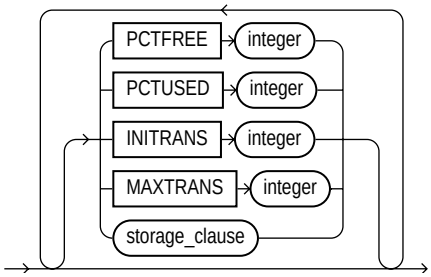
alter_materialized_view::=



(*physical_attributes_clause::=* on page 9-95, *data_segment_compression::=* on page 9-95, *LOB_storage_clause::=* on page 9-95, *modify_LOB_storage_clause::=* on page 9-96, *alter_table_partitioning* on page 11-59 — part of ALTER TABLE syntax, *parallel_*

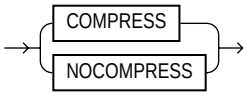
clause::= on page 9-97, *logging_clause::=* on page 9-97, *allocate_extent_clause::=* on page 9-98)

physical_attributes_clause::=

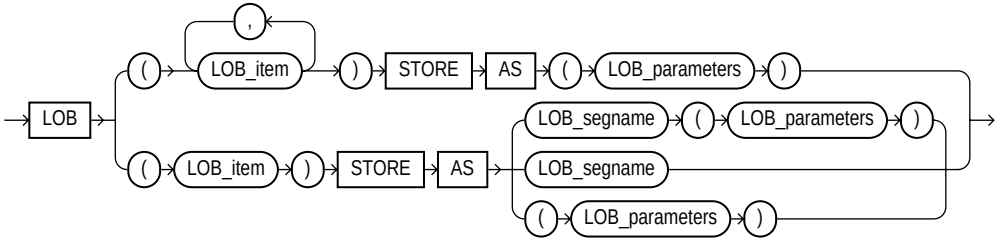


(*storage_clause::=* on page 7-58)

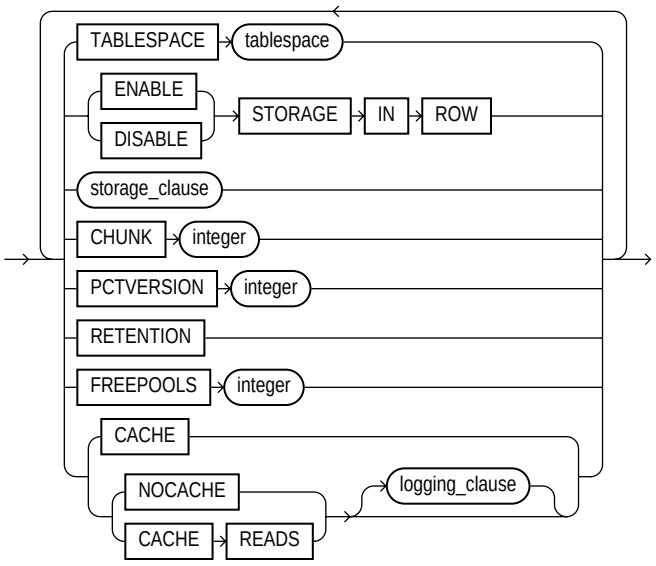
data_segment_compression::=



LOB_storage_clause::=

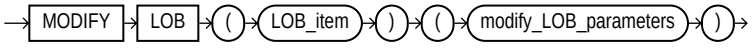


LOB_parameters::=

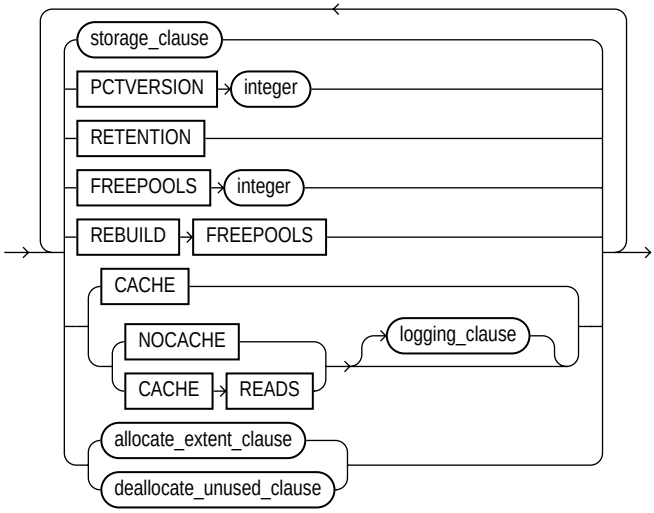


(*storage_clause::=* on page 7-58, *logging_clause::=* on page 7-46)

modify_LOB_storage_clause::=

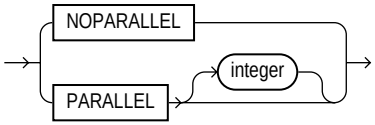


modify_LOB_parameters::=

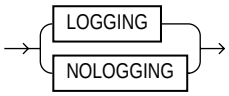


(*storage_clause::=* on page 7-58, *logging_clause::=* on page 7-46, *allocate_extent_clause::=* on page 9-98, *deallocate_unused_clause::=* on page 9-98)

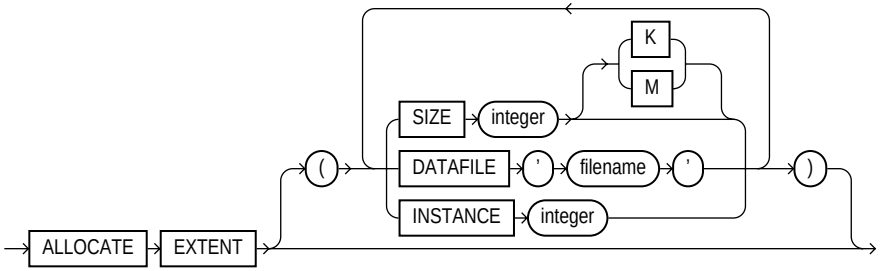
parallel_clause::=



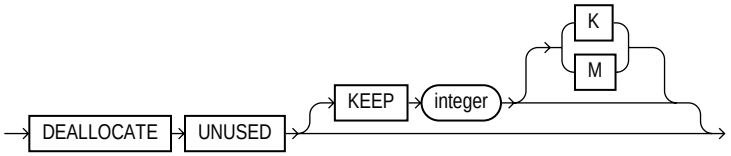
logging_clause::=



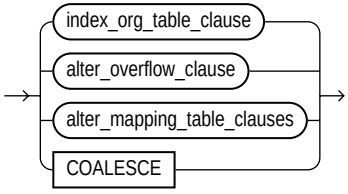
allocate_extent_clause::=



deallocate_unused_clause::=

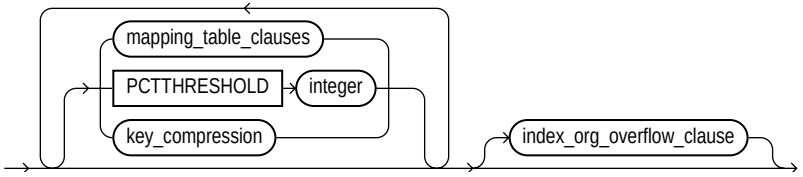


alter_iot_clauses::=



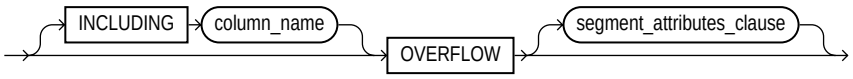
(*index_org_table_clause::=* on page 9-98, *alter_overflow_clause::=* on page 9-99, *alter_mapping_table_clauses*: not supported with materialized views)

index_org_table_clause::=

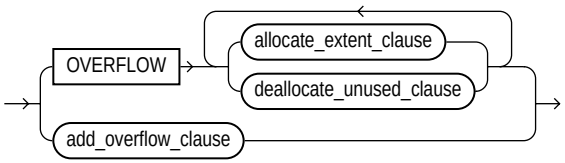


(*mapping_table_clauses*: not supported with materialized views, *key_compression*: not supported with materialized views, *index_org_overflow_clause::=* on page 9-99)

index_org_overflow_clause::=

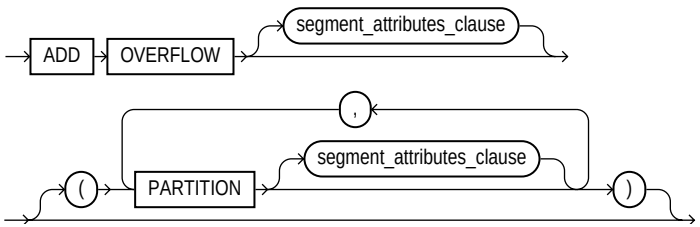


alter_overflow_clause::=

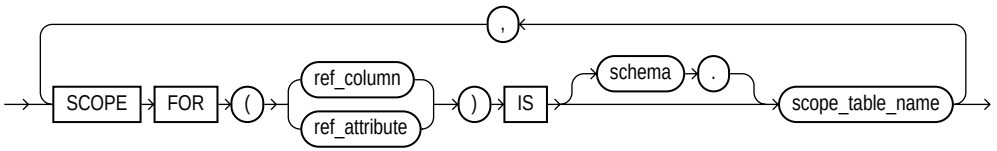


(*allocate_extent_clause::=* on page 9-98, *deallocate_unused_clause::=* on page 9-98)

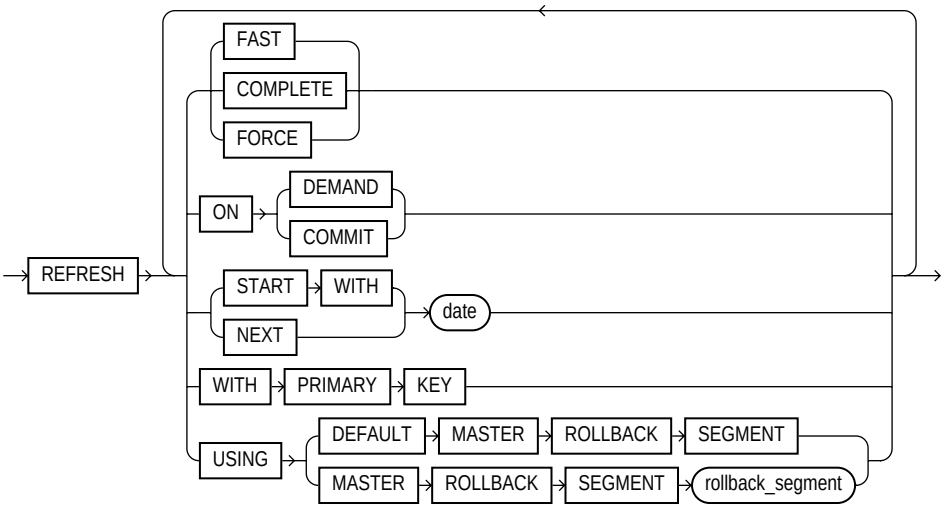
add_overflow_clause::=



scoped_table_ref_constraint::=



alter_mv_refresh::=



Keywords and Parameters

schema

Specify the schema containing the materialized view. If you omit *schema*, Oracle assumes the materialized view is in your own schema.

materialized_view

Specify the name of the materialized view to be altered.

physical_attributes_clause

Specify new values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters (or, when used in the USING INDEX clause, for the INITRANS and MAXTRANS parameters only) and the storage characteristics for the materialized view.

See Also:

- ALTER TABLE on page 11-2 for information on the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters
- *storage_clause* on page 7-56 for information about storage characteristics

data_segment_compression

Use the *data_segment_compression* clause to instruct Oracle whether to compress data segments to reduce disk and memory use. The COMPRESS keyword enables data segment compression. The NOCOMPRESS keyword disables data segment compression.

> **See Also:** *data_segment_compression* clause of CREATE TABLE on page 15-29 for more information on data segment compression

LOB_storage_clause

The *LOB_storage_clause* lets you specify the storage characteristics of a new LOB. LOB storage behaves for materialized views exactly as it does for tables.

> **See Also:** the *LOB_storage_clause* of CREATE TABLE on page 15-37 for information on the LOB storage parameters

modify_LOB_storage_clause

The *modify_LOB_storage_clause* lets you modify the physical attributes of the LOB attribute *lob_item* or LOB object attribute. Modification of LOB storage behaves for materialized views exactly as it does for tables.

> **See Also:** the *modify_LOB_storage_clause* of ALTER TABLE on page 11-56 for information on the LOB storage parameters that can be modified

alter_table_partitioning

The syntax and general functioning of the partitioning clauses for materialized views is the same as for partitioned tables.

> **See Also:** *alter_table_partitioning* on page 11-59

Restrictions on *partitioning_clauses*:

- You cannot specify the *LOB_storage_clause* or *modify_LOB_storage_clause* within any of the *partitioning_clauses*.
- If you attempt to drop, truncate, or exchange a materialized view partition, Oracle raises an error.

Note: If you wish to keep the contents of the materialized view synchronized with those of the master table, Oracle Corporation recommends that you manually perform a complete refresh of all materialized views dependent on the table after dropping or truncating a table partition.

MODIFY PARTITION UNUSABLE LOCAL INDEXES Use this clause to mark UNUSABLE all the local index partitions associated with *partition*.

MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES Use this clause to rebuild the unusable local index partitions associated with *partition*.

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for the materialized view.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

logging_clause

Specify or change the logging characteristics of the materialized view.

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

allocate_extent_clause

The *allocate_extent_clause* lets you explicitly allocate a new extent for the materialized view.

> **See Also:** *allocate_extent_clause* on page 7-2 for a full description of this clause

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

> **See Also:** ALTER TABLE on page 11-2 for information about specifying CACHE or NOCACHE

alter_iot_clauses

Use the *alter_iot_clauses* to change the characteristics of an index-organized materialized view. The keywords and parameters of the components of the *alter_iot_clauses* have the same semantics as in ALTER TABLE, with the restrictions that follow.

Restrictions on *alter_iot_clauses*: You cannot specify the *mapping_table_clauses* or the *key_compression* clause of the *index_org_table_clause*.

> **See Also:** *"index_org_table_clause"* of CREATE MATERIALIZED VIEW on page 14-15 for information on creating an index-organized materialized view

USING INDEX Clause

Use this clause to change the value of INITRANS, MAXTRANS, and STORAGE parameters for the index Oracle uses to maintain the materialized view's data.

Restriction on the USING INDEX clause: You cannot specify the PCTUSED or PCTFREE parameters in this clause.

MODIFY *scoped_table_ref_constraint*

Use the MODIFY *scoped_table_ref_constraint* clause to rescope a REF column or attribute to a new table.

Restrictions on rescoping REF columns: You can rescope only one REF column or attribute in each ALTER MATERIALIZED VIEW statement, and this must be the only clause in this statement.

REBUILD Clause

Specify REBUILD to regenerate refresh operations if a type that is referenced in *materialized_view* has evolved.

Restriction on the REBUILD clause: You cannot specify any other clause in the same ALTER MATERIALIZED VIEW statement.

alter_mv_refresh

Use the *alter_mv_refresh* to change the default method and mode and the default times for automatic refreshes. If the contents of a materialized view's master tables are modified, the data in the materialized view must be updated to make the materialized view accurately reflect the data currently in its master table(s). This clause lets you schedule the times and specify the method and mode for Oracle to refresh the materialized view.

Note: This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle9i Replication* and *Oracle9i Data Warehousing Guide*.

FAST Clause

Specify FAST for incremental refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes are stored either in the materialized view log associated with the master table (for conventional DML changes) or in the direct loader log (for direct-path INSERT operations).

For both conventional DML changes and for direct-path INSERTs, other conditions may restrict the eligibility of a materialized view for fast refresh.

See Also:

- *Oracle9i Replication* for restrictions on fast refresh in replication environments
- *Oracle9i Data Warehousing Guide* for restrictions on fast refresh in data warehouse environments
- "Automatic Refresh: Examples" on page 9-109

Restrictions on FAST refresh:

- When you specify FAST refresh at create time, Oracle verifies that the materialized view you are creating is eligible for fast refresh. When you change the refresh method to FAST in an ALTER MATERIALIZED VIEW statement, Oracle does not perform this verification. If the materialized view is not eligible for fast refresh, Oracle will return an error when you attempt to refresh this view.
- Materialized views are not eligible for fast refresh if the defining query contains an analytic function.

See Also: "Analytic Functions" on page 6-9

COMPLETE Clause

Specify COMPLETE for the complete refresh method, which is implemented by executing the materialized view's defining query. If you request a complete refresh, Oracle performs a complete refresh even if a fast refresh is possible.

See Also: "Complete Refresh: Example" on page 9-110

FORCE Clause

Specify FORCE if, when a refresh occurs, you want Oracle to perform a fast refresh if one is possible or a complete refresh otherwise.

ON COMMIT Clause

Specify ON COMMIT if you want a fast refresh to occur whenever Oracle commits a transaction that operates on a master table of the materialized view.

Restriction on the ON COMMIT clause: This clause is supported only for materialized join views and single-table materialized aggregate views.

See Also: *Oracle9i Replication* and *Oracle9i Data Warehousing Guide*

ON DEMAND Clause

Specify ON DEMAND if you want the materialized view to be refreshed on demand by calling one of the three DBMS_MVIEW refresh procedures. If you omit both ON COMMIT and ON DEMAND, ON DEMAND is the default.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on these procedures
- *Oracle9i Data Warehousing Guide* on the types of materialized views you can create by specifying REFRESH ON DEMAND

Note: If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.

START WITH Clause

Specify START WITH *date* to indicate a date for the first automatic refresh time.

NEXT Clause

Specify NEXT to indicate a date expression for calculating the interval between automatic refreshes.

Both the START WITH and NEXT values must evaluate to a time in the future. If you omit the START WITH value, Oracle determines the first automatic refresh time by evaluating the NEXT expression with respect to the creation time of the materialized view. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the materialized view only once. If you omit both the START WITH and NEXT values, or if you omit the *alter_mv_refresh* entirely, Oracle does not automatically refresh the materialized view.

WITH PRIMARY KEY Clause

Specify WITH PRIMARY KEY to change a rowid materialized view to a primary key materialized view. Primary key materialized views allow materialized view master tables to be reorganized without affecting the materialized view's ability to continue to fast refresh.

For you to specify this clause, the master table must contain an enabled primary key constraint and must have defined on it a materialized view log that logs primary key information.

See Also:

- *Oracle9i Replication* for detailed information about primary key materialized views and *Oracle9i Database Migration* for information on changing rowid materialized views to primary key materialized views
- "Primary Key Materialized View: Example" on page 9-111

USING ROLLBACK SEGMENT Clause

Specify `USING ROLLBACK SEGMENT` to change the remote rollback segment to be used during materialized view refresh, where *`rollback_segment`* is the name of the rollback segment to be used.

See Also: *Oracle9i Replication* for information on changing the local materialized view rollback segment using the `DBMS_REFRESH` package and "Changing Materialized View Rollback Segments: Examples" on page 9-111

DEFAULT Specify `DEFAULT` if you want Oracle to choose the rollback segment to use. If you specify `DEFAULT`, you cannot specify *`rollback_segment`*.

MASTER ... *rollback_segment* Specify the remote rollback segment to be used at the remote master for the individual materialized view. (To change the local materialized view rollback segment, use the `DBMS_REFRESH` package, described in *Oracle9i Replication*.)

One master rollback segment is stored for each materialized view and is validated during materialized view creation and refresh. If the materialized view is complex, the master rollback segment, if specified, is ignored.

QUERY REWRITE Clause

Use this clause to determine whether the materialized view is eligible to be used for query rewrite.

ENABLE Clause

Specify `ENABLE` to enable the materialized view for query rewrite.

See Also: "Enabling Query Rewrite: Example" on page 9-110

Restrictions on the ENABLE clause:

- If the materialized view is in an invalid or unusable state, it is not eligible for query rewrite in spite of the ENABLE mode.
- You cannot enable query rewrite if the materialized view was created totally or in part from a view.
- You can enable query rewrite only if all user-defined functions in the materialized view are DETERMINISTIC.

  See Also: CREATE FUNCTION on page 13-49

- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include CURRENT_TIME or USER.

  See Also: *Oracle9i Data Warehousing Guide* for more information on query rewrite

DISABLE Clause

Specify DISABLE if you do not want the materialized view to be eligible for use by query rewrite. (If a materialized view is in the invalid state, it is not eligible for use by query rewrite, whether or not it is disabled.) However, a disabled materialized view can be refreshed.

COMPILE

Specify COMPILE to explicitly revalidate a materialized view. If an object upon which the materialized view depends is dropped or altered, the materialized view remains accessible, but it is invalid for query rewrite. You can use this clause to explicitly revalidate the materialized view to make it eligible for query rewrite.

If the materialized view fails to revalidate, it cannot be refreshed or used for query rewrite.

See Also: "Compiling a Materialized View: Example" on page 9-111

CONSIDER FRESH

This clause lets you manage the staleness state of a materialized after changes have been made to its master tables. CONSIDER FRESH directs Oracle to consider the materialized view fresh and therefore eligible for query rewrite in the TRUSTED or STALE_TOLERATED modes. Because Oracle cannot guarantee the freshness of the

materialized view, query rewrite in ENFORCED mode is not supported. This clause also sets the staleness state of the materialized view to UNKNOWN. The staleness state is displayed in the STALENESS column of the ALL_MVIEWS, DBA_MVIEWS, and USER_MVIEWS data dictionary views.

Note: A materialized view is stale if changes have been made to the contents of any of its master tables. This clause directs Oracle to assume that the materialized view is fresh and that no such changes have been made. Therefore, actual updates to those tables pending refresh are purged with respect to the materialized view.

See Also: *Oracle9i Data Warehousing Guide* for more information on query rewrite and the implications of performing partition maintenance operations on master tables, and "CONSIDER FRESH: Example" on page 9-110

Examples

Automatic Refresh: Examples The following statement changes the default refresh method for the sales_by_month_by_state materialized view (created in "Creating Materialized Aggregate Views: Example" on page 14-27) to FAST:

```
ALTER MATERIALIZED VIEW sales_by_month_by_state
   REFRESH FAST;
```

The next automatic refresh of the materialized view will be a fast refresh provided it is a simple materialized view and its master table has a materialized view log that was created before the materialized view was created or last refreshed.

Because the REFRESH clause does not specify START WITH or NEXT values, Oracle will use the refresh intervals established by the REFRESH clause when the sales_by_month_by_state materialized view was created or last altered.

The following statement stores a new interval between automatic refreshes for the sales_by_month_by_state materialized view:

```
ALTER MATERIALIZED VIEW sales_by_month_by_state
   REFRESH NEXT SYSDATE+7;
```

Because the REFRESH clause does not specify a START WITH value, the next automatic refresh occurs at the time established by the START WITH and NEXT

values specified when the sales_by_month_by_state materialized view was created or last altered.

At the time of the next automatic refresh, Oracle refreshes the materialized view, evaluates the NEXT expression SYSDATE+7 to determine the next automatic refresh time, and continues to refresh the materialized view automatically once a week. Because the REFRESH clause does not explicitly specify a refresh method, Oracle continues to use the refresh method specified by the REFRESH clause of the CREATE MATERIALIZED VIEW or most recent ALTER MATERIALIZED VIEW statement.

CONSIDER FRESH: Example The following statement instructs Oracle that materialized view sales_by_month_by_state should be considered fresh. This statement allows sales_by_month_by_state to be eligible for query rewrite in TRUSTED mode even after you have performed partition maintenance operations on the master tables of sales_by_month_by_state:

```
ALTER MATERIALIZED VIEW sales_by_month_by_state CONSIDER FRESH;
```

> **See Also:** Splitting Table Partitions: Examples on page 11-92 for a partitioning maintenance example that would require this ALTER MATERIALIZED VIEW example

Complete Refresh: Example The following statement specifies a new refresh method, a new NEXT refresh time, and a new interval between automatic refreshes of the emp_data materialized view (created in "Periodic Refresh of Materialized Views: Example" on page 14-29):

```
ALTER MATERIALIZED VIEW emp_data
   REFRESH COMPLETE
   START WITH TRUNC(SYSDATE+1) + 9/24
   NEXT SYSDATE+7;
```

The START WITH value establishes the next automatic refresh for the materialized view to be 9:00 a.m. tomorrow. At that point, Oracle performs a complete refresh of the materialized view, evaluates the NEXT expression, and subsequently refreshes the materialized view every week.

Enabling Query Rewrite: Example The following statement enables query rewrite on the materialized view mv1 and implicitly revalidates it:

```
ALTER MATERIALIZED VIEW emp_data
   ENABLE QUERY REWRITE;
```

Changing Materialized View Rollback Segments: Examples The following statement changes the remote master rollback segment used during materialized view refresh to `rbs_two`:

```
ALTER MATERIALIZED VIEW new_employees
   REFRESH USING MASTER ROLLBACK SEGMENT rbs_two;
```

The following statement changes the remote master rollback segment used during materialized view refresh to one chosen by Oracle:

```
ALTER MATERIALIZED VIEW new_employees
   REFRESH USING DEFAULT MASTER ROLLBACK SEGMENT;
```

Primary Key Materialized View: Example The following statement changes the rowid materialized view order_data (created in "Creating Rowid Materialized Views: Example" on page 14-29) to a primary key materialized view:

```
ALTER MATERIALIZED VIEW order_data
   REFRESH WITH PRIMARY KEY;
```

Compiling a Materialized View: Example The following statement revalidates the materialized view `store_mv`:

```
ALTER MATERIALIZED VIEW order_data COMPILE;
```

ALTER MATERIALIZED VIEW LOG

Purpose

Use the ALTER MATERIALIZED VIEW LOG statement to alter the storage characteristics, refresh mode or time, or type of an existing materialized view log. A **materialized view log** is a table associated with the master table of a materialized view.

Note: The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

See Also:

- CREATE MATERIALIZED VIEW LOG on page 14-32 for information on creating a materialized view log
- ALTER MATERIALIZED VIEW on page 9-92 for more information on materialized views, including refreshing them
- CREATE MATERIALIZED VIEW on page 14-5 for a description of the various types of materialized views

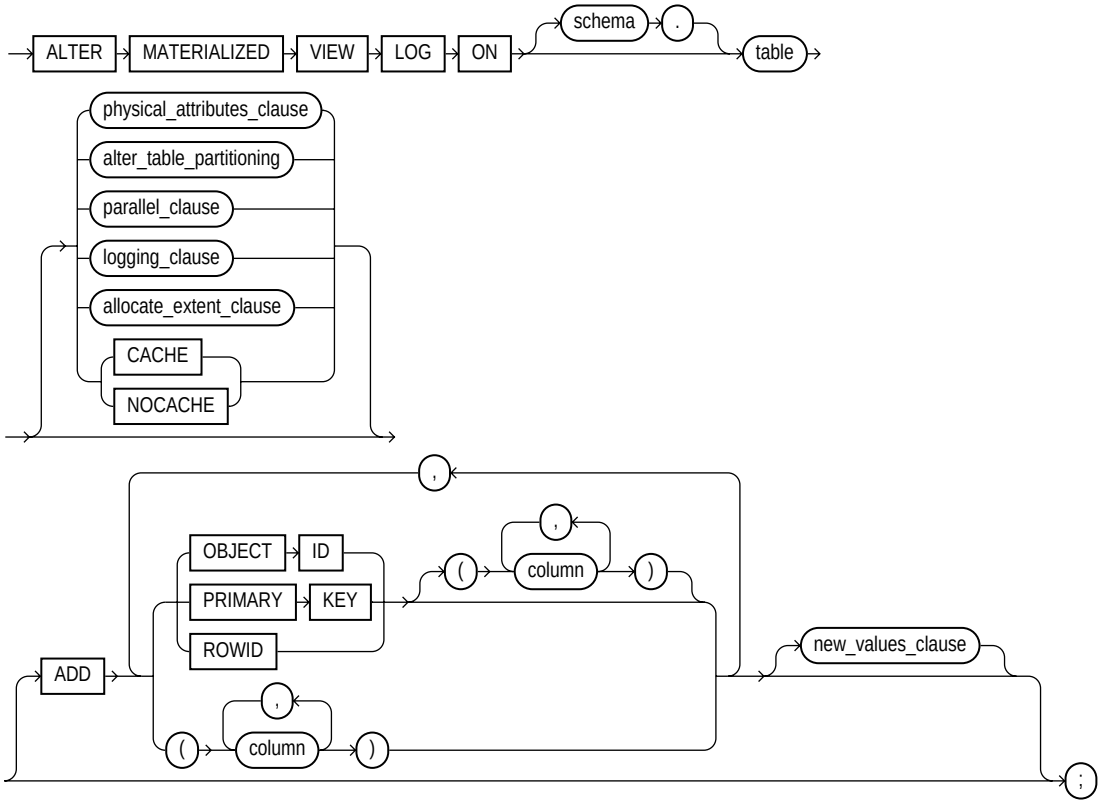
Prerequisites

Only the owner of the master table or a user with the SELECT privilege on the master table and the ALTER privilege on the materialized view log can alter a materialized view log.

See Also: *Oracle9i Replication* for detailed information about the prerequisites for ALTER MATERIALIZED VIEW LOG

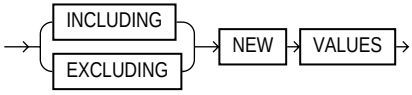
Syntax

alter_materialized_view_log::=

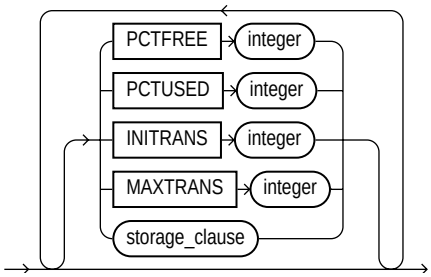


(*physical_attributes_clause::=* on page 9-114, *alter_table_partitioning* on page 11-59 — part of ALTER TABLE syntax, *parallel_clause::=* on page 9-114, *logging_clause::=* on page 7-46, *allocate_extent_clause::=* on page 9-114, *new_values_clause::=* on page 9-113),

new_values_clause::=

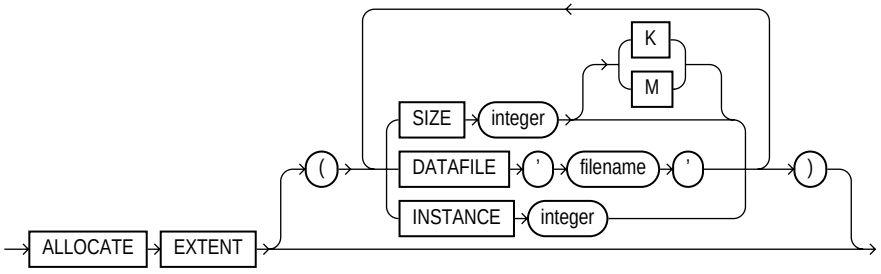


physical_attributes_clause::=

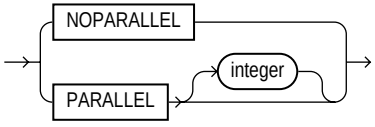


(*storage_clause::=* on page 7-58)

allocate_extent_clause::=



parallel_clause::=



Keywords and Parameters

schema

Specify the schema containing the master table. If you omit *schema*, Oracle assumes the materialized view log is in your own schema.

table

Specify the name of the master table associated with the materialized view log to be altered.

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and storage characteristics for the table, the partition, the overflow data segment, or the default characteristics of a partitioned table.

Restriction on the *physical_attributes_clause*: You cannot use the *storage_clause* to modify extent parameters if the materialized view log resides in a locally managed tablespace.

> **See Also:** CREATE TABLE on page 15-7 for a description of these parameters

alter_table_partitioning

The syntax and general functioning of the partitioning clauses is the same as described for the ALTER TABLE statement.

Restrictions on *partitioning_clauses*:

- You cannot use the *LOB_storage_clause* or *modify_LOB_storage_clause* when modifying partitions of a materialized view log.
- If you attempt to drop, truncate, or exchange a materialized view log partition, Oracle raises an error.

> **See Also:** *alter_table_partitioning* on page 11-59

parallel_clause

The *parallel_clause* lets you specify whether parallel operations will be supported for the materialized view log.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

logging_clause

Specify the logging attribute of the materialized view log.

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

allocate_extent_clause

Use the *allocate_extent_clause* to explicitly allocate a new extent for the materialized view log.

> **See Also:** *allocate_extent_clause* on page 7-2 for a full description of this clause

CACHE | NOCACHE Clause

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this log are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the least recently used end of the LRU list.

> **See Also:** CREATE TABLE on page 15-7 for information about specifying CACHE or NOCACHE

ADD Clause

The ADD clause lets you augment the materialized view log so that it records the primary key values, rowid values, or object ID values when rows in the materialized view master table are changed. This clause can also be used to record additional columns.

To stop recording any of this information, you must first drop the materialized view log and then re-create it. Dropping the materialized view log and then re-creating it

forces each of the existing materialized views that depend on the master table to complete refresh on its next refresh.

Restriction on the ADD clause: You can specify only one PRIMARY KEY, one ROWID, one OBJECT ID and one column list for each materialized view log. Therefore, if any of these three values were specified at create time (either implicitly or explicitly), you cannot specify those values in this ALTER statement.

OBJECT ID Specify OBJECT ID if you want the appropriate object identifier of all rows that are changed to be recorded in the materialized view log.

Restriction on the OBJECT ID clause: You can specify OBJECT ID only for logs on object tables, and you cannot specify it for storage tables.

PRIMARY KEY Specify PRIMARY KEY if you want the primary key values of all rows that are changed to be recorded in the materialized view log.

ROWID Specify ROWID if you want the rowid values of all rows that are changed to be recorded in the materialized view log.

column Specify the additional columns whose values you want to be recorded in the materialized view log for all rows that are changed. Typically these columns are filter columns (non-primary-key columns referenced by materialized views) and join columns (non-primary-key columns that define a join in the WHERE clause of the subquery).

See Also:

- CREATE MATERIALIZED VIEW on page 14-5 for details on explicit and implicit inclusion of materialized view log values
- *Oracle9i Replication* for more information about filter columns and join columns
- "Rowid Materialized View Log: Example" on page 9-118

NEW VALUES Clause

The NEW VALUES clause lets you specify whether Oracle saves both old and new values in the materialized view log. The value you set in this clause applies to all columns in the log, not only to primary key, rowid, or columns you may have added in this ALTER MATERIALIZED VIEW LOG statement.

INCLUDING Specify INCLUDING to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, you must specify INCLUDING.

EXCLUDING Specify EXCLUDING to disable the recording of new values in the log. You can use this clause to avoid the overhead of recording new values.

If you have a fast-refreshable single-table materialized aggregate view defined on this table, do not specify EXCLUDING NEW VALUES unless you first change the refresh mode of the materialized view to something other than FAST.

See Also: "Materialized View Log EXCLUDING NEW VALUES: Example" on page 9-118

Examples

Rowid Materialized View Log: Example The following statement alters an existing primary key materialized view log to also record rowid information:

```
ALTER MATERIALIZED VIEW LOG ON order_items ADD ROWID;
```

Materialized View Log EXCLUDING NEW VALUES: Example The following statement alters the materialized view log on hr.employees by adding a filter column and excluding new values. Any materialized aggregate views that use this log will no longer be fast refreshable. However, if fast refresh is no longer needed, this action avoids the overhead of recording new values:

```
ALTER MATERIALIZED VIEW LOG ON employees
   ADD (commission_pct)
   EXCLUDING NEW VALUES;
```

ALTER OPERATOR

Purpose

Use the ALTER OPERATOR statement to compile an existing operator.

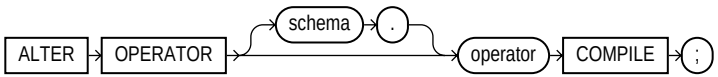
See Also: CREATE OPERATOR on page 14-42

Prerequisites

The operator must be in your own or another schema, or you must have the ALTER ANY OPERATOR system privilege.

Syntax

alter_operator::=



Keywords and Parameters

schema

Specify the schema containing the operator. If you omit this clause, Oracle assumes the operator is in your own schema.

operator

Specify the name of the operator to be recompiled.

COMPILE

Specify COMPILE to cause Oracle to recompile the operator. The COMPILE keyword is required.

Examples

Compiling a User-defined Operator: Example The following example compiles the operator eq_op (which was created in "Creating User-Defined Operators: Example" on page 14-45):

```
ALTER OPERATOR eq_op COMPILE;
```

ALTER OUTLINE

Purpose

Use the ALTER OUTLINE statement to rename a stored outline, reassign it to a different category, or regenerate it by compiling the outline's SQL statement and replacing the old outline data with the outline created under current conditions.

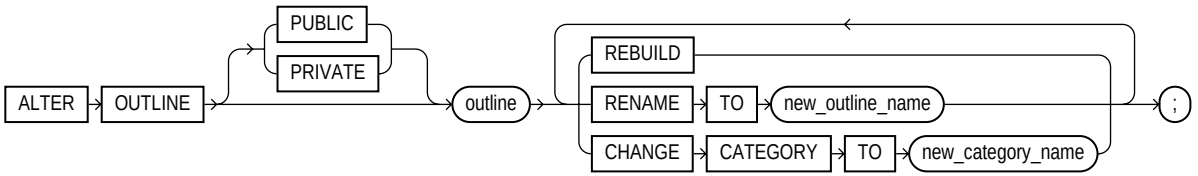
> **See Also:** CREATE OUTLINE on page 14-46 and *Oracle9i Database Performance Tuning Guide and Reference* for more information on outlines

Prerequisites

To modify an outline, you must have the ALTER ANY OUTLINE system privilege.

Syntax

alter_outline::=



Keywords and Parameters

PUBLIC | PRIVATE

Specify PUBLIC if you want to modify the public version of this outline. This is the default.

Specify PRIVATE if you want to modify the outline that is private to the current session and whose data is stored in the current parsing schema.

outline

Specify the name of the outline to be modified.

REBUILD

Specify REBUILD to regenerate the execution plan for *outline* using current conditions.

See Also: "Rebuilding an Outline: Example" on page 9-121

RENAME TO Clause

Use the RENAME TO clause to specify an outline name to replace *outline*.

CHANGE CATEGORY TO Clause

Use the CHANGE CATEGORY TO clause to specify the name of the category into which the *outline* will be moved.

Example

Rebuilding an Outline: Example The following statement regenerates a stored outline called salaries by compiling the outline's text and replacing the old outline data with the outline created under current conditions.

```
ALTER OUTLINE salaries REBUILD;
```

ALTER PACKAGE

Purpose

Use the ALTER PACKAGE statement to explicitly recompile a package specification, body, or both. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects together. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

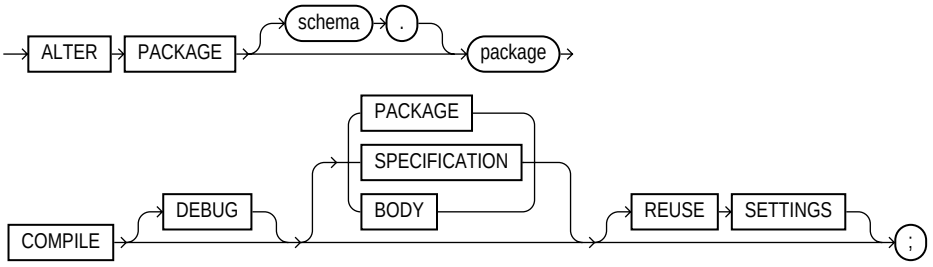
Note: This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the CREATE PACKAGE or the CREATE PACKAGE BODY on page 14-50 statement with the OR REPLACE clause.

Prerequisites

For you to modify a package, the package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

alter_package::=



Keywords and Parameters

schema

Specify the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.

package

Specify the name of the package to be recompiled.

COMPILE

You must specify `COMPILE` to recompile the package specification or body. The `COMPILE` keyword is required.

During recompilation, Oracle drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

If recompiling the package results in compilation errors, Oracle returns an error and the body remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

See Also: "Recompiling a Package: Examples" on page 9-124

SPECIFICATION

Specify `SPECIFICATION` to recompile only the package specification, regardless of whether it is invalid. You might want to recompile a package specification to check for compilation errors after modifying the specification.

When you recompile a package specification, Oracle invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

BODY

Specify `BODY` to recompile only the package body regardless of whether it is invalid. You might want to recompile a package body after modifying it. Recompiling a package body does not invalidate objects that depend upon the package specification.

When you recompile a package body, Oracle first recompiles the objects on which the body depends, if any of those objects are invalid. If Oracle recompiles the body successfully, the body becomes valid.

PACKAGE

Specify PACKAGE to recompile both the package specification and the package body if one exists, regardless of whether they are invalid. This is the default. The recompilation of the package specification and body lead to the invalidation and recompilation as described for SPECIFICATION and BODY.

> **See Also:** *Oracle9i Database Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects

DEBUG

Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on debugging packages

REUSE SETTINGS

Specify REUSE SETTINGS to prevent Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation.

If you specify both DEBUG and REUSE SETTINGS, Oracle sets the persistently stored value of the PLSQL_COMPILER_FLAGS parameter to INTERPRETED, DEBUG. No other compiler switch values are changed.

> **See Also:** *PL/SQL User's Guide and Reference* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on the interaction of the PLSQL_COMPILER_FLAGS parameter with the COMPILE clause

Examples

Recompiling a Package: Examples This statement explicitly recompiles the specification and body of the hr.emp_mgmt package that was created in "Creating a Package: Example" on page 14-53:

```
ALTER PACKAGE emp_mgmt
   COMPILE PACKAGE;
```

If Oracle encounters no compilation errors while recompiling the `accounting` specification and body, `emp_mgmt` becomes valid. The user `hr` can subsequently call or reference all package objects declared in the specification of `emp_mgmt` without run-time recompilation. If recompiling `emp_mgmt` results in compilation errors, Oracle returns an error and `emp_mgmt` remains invalid.

Oracle also invalidates all objects that depend upon `emp_mgmt`. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

To recompile the body of the `emp_mgmt` package in the schema `hr`, issue the following statement:

```
ALTER PACKAGE hr.emp_mgmt
   COMPILE BODY;
```

If Oracle encounters no compilation errors while recompiling the package body, the body becomes valid. The user `hr` can subsequently call or reference all package objects declared in the specification of `emp_mgmt` without run-time recompilation. If recompiling the body results in compilation errors, Oracle returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of `emp_mgmt`, Oracle does not invalidate dependent objects.

ALTER PROCEDURE

Purpose

Use the ALTER PROCEDURE statement to explicitly recompile a standalone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the ALTER PACKAGE statement (see ALTER PACKAGE on page 9-122).

Note: This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a procedure, use the CREATE PROCEDURE statement with the OR REPLACE clause (see CREATE PROCEDURE on page 14-62).

The ALTER PROCEDURE statement is quite similar to the ALTER FUNCTION statement.

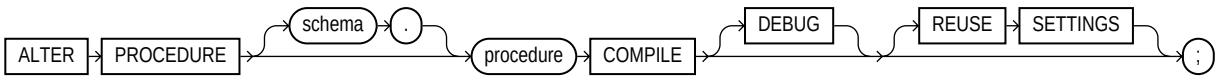
See Also: ALTER FUNCTION on page 9-61

Prerequisites

The procedure must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

alter_procedure::=



Keywords and Parameters

schema

Specify the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.

procedure

Specify the name of the procedure to be recompiled.

COMPILE

Specify `COMPILE` to recompile the procedure. The `COMPILE` keyword is required. Oracle recompiles the procedure regardless of whether it is valid or invalid.

- Oracle first recompiles objects upon which the procedure depends, if any of those objects are invalid.
- Oracle also invalidates any local objects that depend upon the procedure, such as procedures that call the recompiled procedure or package bodies that define procedures that call the recompiled procedure.
- If Oracle recompiles the procedure successfully, the procedure becomes valid. If recompiling the procedure results in compilation errors, then Oracle returns an error and the procedure remains invalid. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

During recompilation, Oracle drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

> **See Also:** *Oracle9i Database Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects and "Recompiling a Procedure: Example" on page 9-128

DEBUG

Specify `DEBUG` to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for information on debugging procedures

REUSE SETTINGS

Specify `REUSE SETTINGS` to prevent Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation.

If you specify both `DEBUG` and `REUSE SETTINGS`, Oracle sets the persistently stored value of the `PLSQL_COMPILER_FLAGS` parameter to `INTERPRETED, DEBUG`. No other compiler switch values are changed.

See Also: *PL/SQL User's Guide and Reference* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on the interaction of the PLSQL_COMPILER_FLAGS parameter with the COMPILE clause

Example

Recompiling a Procedure: Example To explicitly recompile the procedure remove_emp owned by the user hr, issue the following statement:

```
ALTER PROCEDURE hr.remove_emp
   COMPILE;
```

If Oracle encounters no compilation errors while recompiling credit, credit becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling credit results in compilation errors, Oracle returns an error and credit remains invalid.

Oracle also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that call credit. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

ALTER PROFILE

Purpose

Use the ALTER PROFILE statement to add, modify, or remove a resource limit or password management parameter in a profile.

Changes made to a profile with an ALTER PROFILE statement affect users only in their subsequent sessions, not in their current sessions.

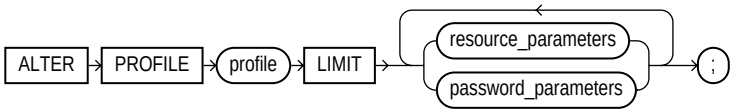
> **See Also:** CREATE PROFILE on page 14-69 for information on creating a profile

Prerequisites

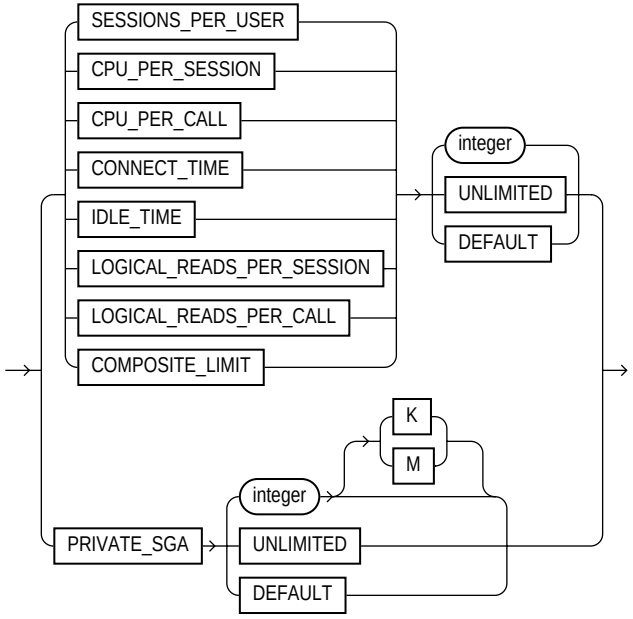
You must have ALTER PROFILE system privilege to change profile resource limits. To modify password limits and protection, you must have ALTER PROFILE and ALTER USER system privileges.

Syntax

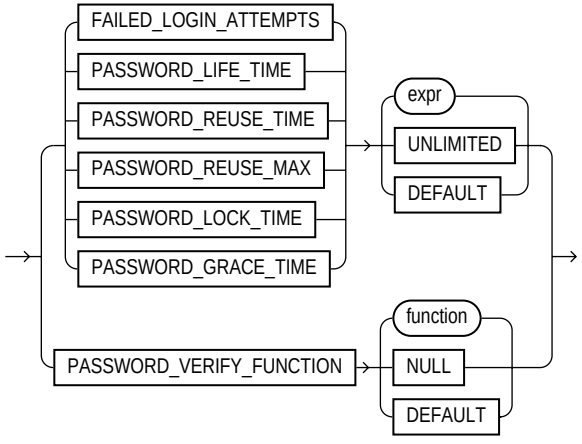
alter_profile::=



resource_parameters::=



password_parameters::=



Keywords and Parameters

The keywords and parameters in the ALTER PROFILE statement all have the same meaning as in the CREATE PROFILE statement.

Note: You cannot remove a limit from the DEFAULT profile.

See Also: CREATE PROFILE on page 14-69 and the examples in the next section

Examples

Making a Password Unavailable: Example The following statement makes the password of the new_profile profile (created in "Creating a Profile: Example" on page 14-74) unavailable for reuse for 90 days:

```
ALTER PROFILE new_profile
   LIMIT PASSWORD_REUSE_TIME 90
   PASSWORD_REUSE_MAX UNLIMITED;
```

Setting Default Password Values: Example The following statement defaults the PASSWORD_REUSE_TIME value of the app_user profile (created in "Setting Profile Password Limits: Example" on page 14-75) to its defined value in the DEFAULT profile:

```
ALTER PROFILE app_user
   LIMIT PASSWORD_REUSE_TIME DEFAULT
   PASSWORD_REUSE_MAX UNLIMITED;
```

Limiting Login Attempts and Password Lock Time: Example The following statement alters profile app_user with FAILED_LOGIN_ATTEMPTS set to 5 and PASSWORD_LOCK_TIME set to 1:

```
ALTER PROFILE app_user LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LOCK_TIME 1;
```

This statement causes app_user's account to become locked for 1 day after 5 unsuccessful login attempts.

Changing Password Lifetime and Grace Period: Example The following statement modifies profile app_user2 PASSWORD_LIFE_TIME to 90 days and PASSWORD_GRACE_TIME to 5 days:

```
ALTER PROFILE app_user2 LIMIT
   PASSWORD_LIFE_TIME 90
   PASSWORD_GRACE_TIME 5;
```

Limiting Concurrent Sessions: Example This statement defines a new limit of 5 concurrent sessions for the app_user profile:

```
ALTER PROFILE app_user LIMIT SESSIONS_PER_USER 5;
```

If the engineer profile does not currently define a limit for SESSIONS_PER_USER, the preceding statement adds the limit of 5 to the profile. If the profile already defines a limit, the preceding statement redefines it to 5. Any user assigned the engineer profile is subsequently limited to 5 concurrent sessions.

Removing Profile Limits: Example This statement removes the IDLE_TIME limit from the app_user profile:

```
ALTER PROFILE app_user LIMIT IDLE_TIME DEFAULT;
```

Any user assigned the app_user profile is subject in their subsequent sessions to the IDLE_TIME limit defined in the DEFAULT profile.

Limiting Profile Idle Time: Example This statement defines a limit of 2 minutes of idle time for the DEFAULT profile:

```
ALTER PROFILE default LIMIT IDLE_TIME  2;
```

This IDLE_TIME limit applies to these users:

- Users who are not explicitly assigned any profile
- Users who are explicitly assigned a profile that does not define an IDLE_TIME limit

This statement defines unlimited idle time for the app_user2 profile:

```
ALTER PROFILE app_user2 LIMIT IDLE_TIME UNLIMITED;
```

Any user assigned the app_user2 profile is subsequently permitted unlimited idle time.

ALTER RESOURCE COST

Purpose

Use the ALTER RESOURCE COST statement to specify or change the formula by which Oracle calculates the total resource cost used in a session.

Although Oracle monitors the use of other resources, only the four resources shown in the syntax can contribute to the total resource cost for a session.

Once you have specified a formula for the total resource cost, you can limit this cost for a session with the COMPOSITE_LIMIT parameter of the CREATE PROFILE statement. If a session's cost exceeds the limit, Oracle aborts the session and returns an error. If you use the ALTER RESOURCE COST statement to change the weight assigned to each resource, Oracle uses these new weights to calculate the total resource cost for all current and subsequent sessions.

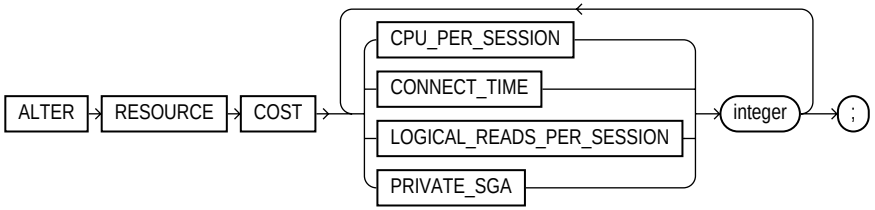
> **See Also:** CREATE PROFILE on page 14-69 for information on all resources and on establishing resource limits

Prerequisites

You must have ALTER RESOURCE COST system privilege.

Syntax

alter_resource_cost::=



Keywords and Parameters

CPU_PER_SESSION

Specify the amount of CPU time that can be used by a session measured in hundredth of seconds.

CONNECT_TIME

Specify the elapsed time allowed for a session measured in minutes.

LOGICAL_READS_PER_SESSION

Specify the number of data blocks that can be read during a session, including blocks read from both memory and disk.

PRIVATE_SGA

Specify the number of bytes of private space in the system global area (SGA) that can be used by a session. This limit applies only if you are using Shared Server architecture and allocating private space in the SGA for your session.

integer

Specify the weight of each resource. The weight that you assign to each resource determines how much the use of that resource contributes to the total resource cost. If you do not assign a weight to a resource, the weight defaults to 0, and use of the resource subsequently does not contribute to the cost. The weights you assign apply to all subsequent sessions in the database.

Oracle calculates the total resource cost by first multiplying the amount of each resource used in the session by the resource's weight, and then summing the products for all four resources. For any session, this cost is limited by the value of the `COMPOSITE_LIMIT` parameter in the user's profile. Both the products and the total cost are expressed in units called **service units**.

Example

Altering Resource Costs: Examples The following statement assigns weights to the resources `CPU_PER_SESSION` and `CONNECT_TIME`:

```
ALTER RESOURCE COST
   CPU_PER_SESSION 100
   CONNECT_TIME      1;
```

The weights establish this cost formula for a session:

```
cost = (100 * CPU_PER_SESSION) + (1 * CONNECT_TIME)
```

where the values of `CPU_PER_SESSION` and `CONNECT_TIME` are either values in the `DEFAULT` profile or in the profile of the user of the session.

Because the preceding statement assigns no weight to the resources LOGICAL_READS_PER_SESSION and PRIVATE_SGA, these resources do not appear in the formula.

If a user is assigned a profile with a COMPOSITE_LIMIT value of 500, a session exceeds this limit whenever cost exceeds 500. For example, a session using 0.04 seconds of CPU time and 101 minutes of elapsed time exceeds the limit. A session using 0.0301 seconds of CPU time and 200 minutes of elapsed time also exceeds the limit.

You can subsequently change the weights with another ALTER RESOURCE statement:

```
ALTER RESOURCE COST
   LOGICAL_READS_PER_SESSION 2
   CONNECT_TIME 0;
```

These new weights establish a new cost formula:

```
cost = (100 * CPU_PER_SESSION) + (2 * LOGICAL_READ_PER_SECOND)
```

where the values of CPU_PER_SESSION and LOGICAL_READS_PER_SECOND are either the values in the DEFAULT profile or in the profile of the user of this session.

This ALTER RESOURCE COST statement changes the formula in these ways:

- The statement omits a weight for the CPU_PER_SESSION resource and the resource was already assigned a weight, so the resource remains in the formula with its original weight.
- The statement assigns a weight to the LOGICAL_READS_PER_SESSION resource, so this resource now appears in the formula.
- The statement assigns a weight of 0 to the CONNECT_TIME resource, so this resource no longer appears in the formula.
- The statement omits a weight for the PRIVATE_SGA resource and the resource was not already assigned a weight, so the resource still does not appear in the formula.

ALTER ROLE

Purpose

Use the ALTER ROLE statement to change the authorization needed to enable a role.

See Also:

- CREATE ROLE on page 14-77 for information on creating a role
- SET ROLE on page 18-47 for information on enabling or disabling a role for your session

Prerequisites

You must either have been granted the role with the ADMIN OPTION or have ALTER ANY ROLE system privilege.

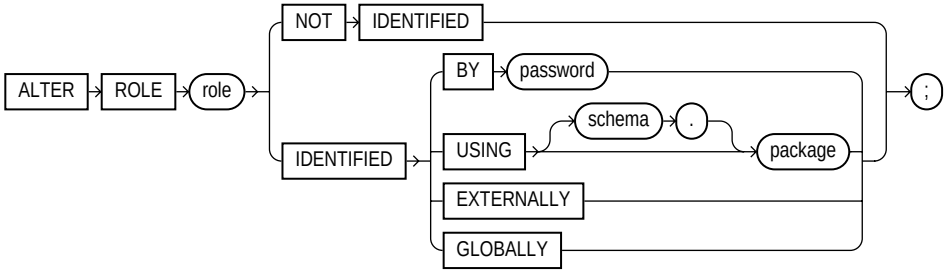
Before you alter a role to IDENTIFIED GLOBALLY, you must:

- Revoke all grants of roles identified externally to the role and
- Revoke the grant of the role from all users, roles, and PUBLIC.

The one exception to this rule is that you should not revoke the role from the user who is currently altering the role.

Syntax

alter_role::=



Keywords and Parameters

The keywords and parameters in the ALTER ROLE statement all have the same meaning as in the CREATE ROLE statement.

Notes:

- When you alter a role, user sessions in which the role is already enabled are not affected.
- If you change a role identified by password to an application role (with the `USING` *`package`* clause), password information associated with the role is lost. Oracle will use the new authentication mechanism the next time the role is to be enabled.
- If you have the `ALTER ANY ROLE` system privilege and you change a role that is `IDENTIFIED GLOBALLY` to `IDENTIFIED BY` *`password`*, `IDENTIFIED EXTERNALLY`, or `NOT IDENTIFIED`, then Oracle grants you the altered role with the `ADMIN OPTION`, as it would have if you had created the role identified nonglobally.

See Also: CREATE ROLE on page 14-77 and the examples that follow

Examples

Changing Role Identification: Example The following statement changes the role `warehouse_user` (created in "Creating a Role: Example" on page 14-79) to `NOT IDENTIFIED`:

```
ALTER ROLE warehouse_user NOT IDENTIFIED;
```

Changing a Role Password: Example This statement changes the password on the `dw_manager` role (created in "Creating a Role: Example" on page 14-79) to `data`:

```
ALTER ROLE dw_manager
   IDENTIFIED BY data;
```

Users granted the `dw_manager` role must subsequently enter the new password "data" to enable the role.

Application Roles: Example The following example changes the `dw_manager` role to an application role using the `hr.admin` package:

```
ALTER ROLE dw_manager IDENTIFIED USING hr.admin;
```

ALTER ROLLBACK SEGMENT

Purpose

Use the ALTER ROLLBACK SEGMENT statement to bring a rollback segment online or offline, to change its storage characteristics, or to shrink it to an optimal or specified size.

The information in this section assumes that your database is running in rollback undo mode (the UNDO_MANAGEMENT initialization parameter is set to MANUAL or not set at all).

If your database is running in Automatic Undo Management mode (the UNDO_MANAGEMENT initialization parameter is set to AUTO), then user-created rollback segments are irrelevant. In this case, Oracle returns an error in response to any CREATE ROLLBACK SEGMENT or ALTER ROLLBACK SEGMENT statement. To suppress these errors, set the UNDO_SUPPRESS_ERRORS parameter to TRUE.

See Also:

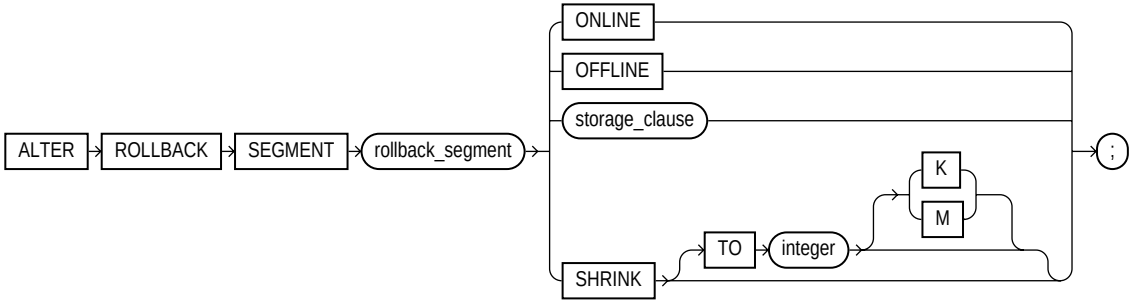
- CREATE ROLLBACK SEGMENT on page 14-80 for information on creating a rollback segment
- *Oracle9i Database Reference* for information on the UNDO_MANAGEMENT and UNDO_SUPPRESS_ERRORS parameters

Prerequisites

You must have ALTER ROLLBACK SEGMENT system privilege.

Syntax

alter_rollback_segment::=



(*storage_clause* on page 7-56)

Keywords and Parameters

rollback_segment

Specify the name of an existing rollback segment.

ONLINE

Specify ONLINE to bring the rollback segment online. When you create a rollback segment, it is initially offline and not available for transactions. This clause brings the rollback segment online, making it available for transactions by your instance. You can also bring a rollback segment online when you start your instance with the initialization parameter ROLLBACK_SEGMENTS.

> **See Also:** "Bringing a Rollback Segment Online: Example" on page 9-140

OFFLINE

Specify OFFLINE to take the rollback segment offline.

- If the rollback segment does not contain any information needed to roll back an active transaction, Oracle takes it offline immediately.
- If the rollback segment does contain information for active transactions, Oracle makes the rollback segment unavailable for future transactions and takes it offline after all the active transactions are committed or rolled back.

Once the rollback segment is offline, it can be brought online by any instance.

To see whether a rollback segment is online or offline, query the data dictionary view DBA_ROLLBACK_SEGS. Online rollback segments have a STATUS value of IN_USE. Offline rollback segments have a STATUS value of AVAILABLE.

Restriction on the OFFLINE clause: You cannot take the SYSTEM rollback segment offline.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information on making rollback segments available and unavailable

storage_clause

Use the *storage_clause* to change the rollback segment's storage characteristics.

Restriction on the *storage_clause*: You cannot change the values of the INITIAL and MINEXTENTS for an existing rollback segment.

> **See Also:** *storage_clause* on page 7-56 for syntax and additional information "Changing Rollback Segment Storage: Example" on page 9-141

SHRINK Clause

Specify SHRINK if you want Oracle to attempt to shrink the rollback segment to an optimal or specified size. The success and amount of shrinkage depend on the available free space in the rollback segment and how active transactions are holding space in the rollback segment.

The value of *integer* is in bytes, unless you specify K or M for kilobytes or megabytes.

If you do not specify TO *integer*, then the size defaults to the OPTIMAL value of the *storage_clause* of the CREATE ROLLBACK SEGMENT statement that created the rollback segment. If OPTIMAL was not specified, then the size defaults to the MINEXTENTS value of the *storage_clause* of the CREATE ROLLBACK SEGMENT statement.

Regardless of whether you specify TO *integer*:

- The value to which Oracle shrinks the rollback segment is valid for the execution of the statement. Thereafter, the size reverts to the OPTIMAL value of the CREATE ROLLBACK SEGMENT statement.
- The rollback segment cannot shrink to less than two extents.

To determine the actual size of a rollback segment after attempting to shrink it, query the BYTES, BLOCKS, and EXTENTS columns of the DBA_SEGMENTS view.

Restriction on the SHRINK clause: In a Real Application Clusters environment, you can shrink only rollback segments that are online to your instance.

> **See Also:** "Resizing a Rollback Segment: Example" on page 9-141

Examples

Bringing a Rollback Segment Online: Example This statement brings the rollback segment rbs_one online:

```
ALTER ROLLBACK SEGMENT rbs_one ONLINE;
```

Changing Rollback Segment Storage: Example This statement changes the STORAGE parameters for rbs_one:

```
ALTER ROLLBACK SEGMENT rbs_one
   STORAGE (NEXT 1000 MAXEXTENTS 20);
```

Resizing a Rollback Segment: Example This statement attempts to resize a rollback segment to 100 megabytes:

```
ALTER ROLLBACK SEGMENT rbs_one
   SHRINK TO 100 M;
```

ALTER SEQUENCE

Purpose

Use the ALTER SEQUENCE statement to change the increment, minimum and maximum values, cached numbers, and behavior of an existing sequence. This statement affects only future sequence numbers.

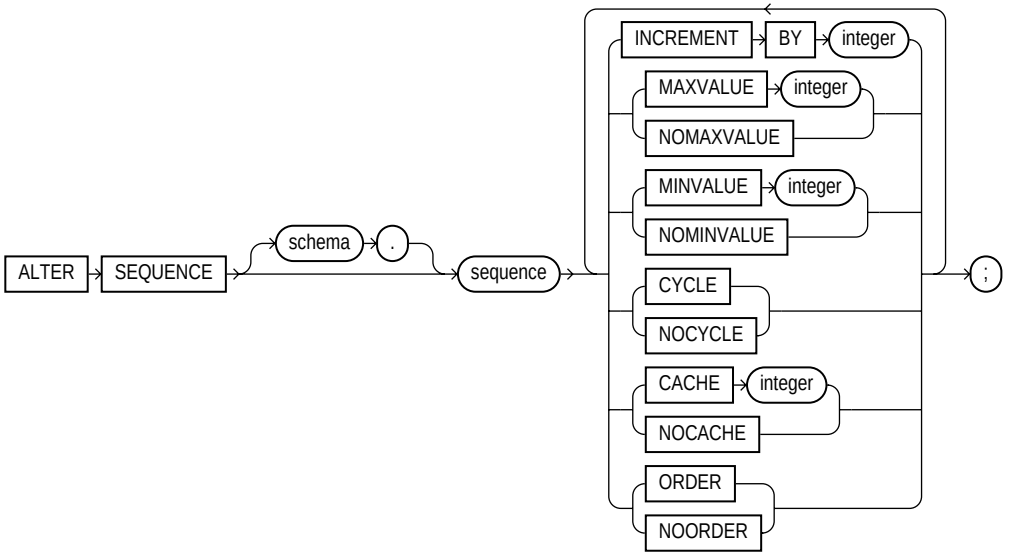
> **See Also:** CREATE SEQUENCE on page 14-87 for additional information on sequences

Prerequisites

The sequence must be in your own schema, or you must have the ALTER object privilege on the sequence, or you must have the ALTER ANY SEQUENCE system privilege.

Syntax

alter_sequence::=



Keywords and Parameters

The keywords and parameters in this statement serve the same purposes they serve when you create a sequence.

- To restart the sequence at a different number, you must drop and re-create it.
- If you change the `INCREMENT BY` value before the first invocation of `NEXTVAL`, some sequence numbers will be skipped. Therefore, if you want to retain the original `START WITH` value, you must drop the sequence and re-create it with the original `START WITH` value and the new `INCREMENT BY` value.
- Oracle performs some validations. For example, a new `MAXVALUE` cannot be imposed that is less than the current sequence number.

See Also:

- CREATE SEQUENCE on page 14-87 for information on creating a sequence
- DROP SEQUENCE on page 17-2 for information on dropping and re-creating a sequence

Examples

Modifying a Sequence: Examples This statement sets a new maximum value for the `customers_seq` sequence:

```
ALTER SEQUENCE customers_seq
   MAXVALUE 1500;
```

This statement turns on `CYCLE` and `CACHE` for the `customers_seq` sequence:

```
ALTER SEQUENCE customers_seq
   CYCLE
   CACHE 5;
```

10

SQL Statements: ALTER SESSION to ALTER SYSTEM

This chapter contains the following SQL statements:

- ALTER SESSION
- ALTER SYSTEM

ALTER SESSION

Purpose

Use the ALTER SESSION statement to specify or modify any of the conditions or parameters that affect your connection to the database. The statement stays in effect until you disconnect from the database.

Prerequisites

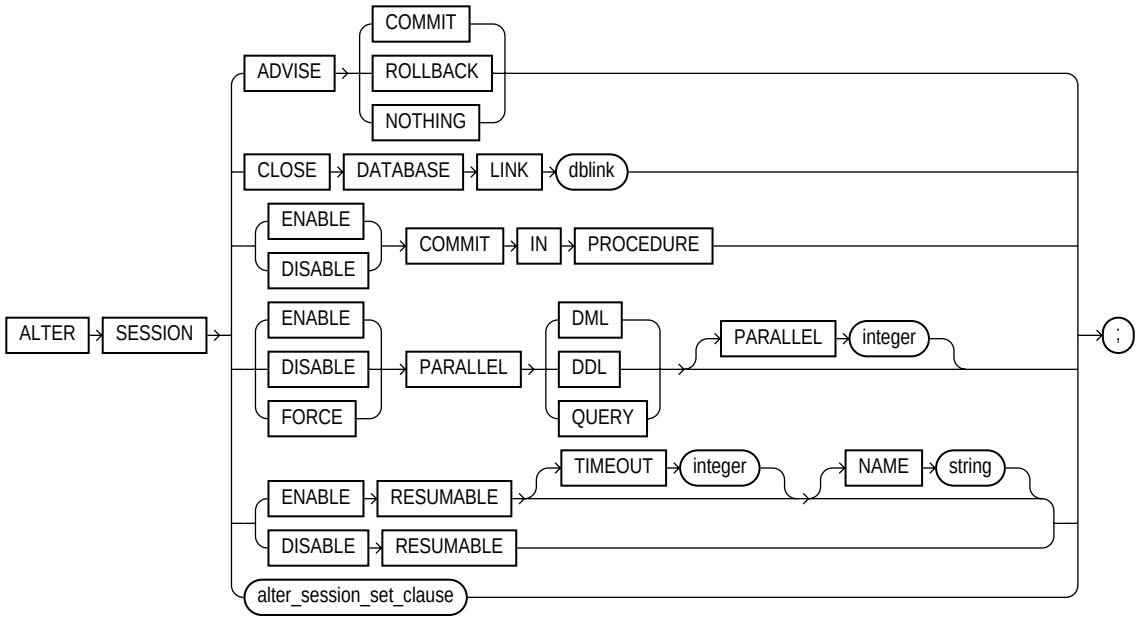
To enable and disable the SQL trace facility, you must have ALTER SESSION system privilege.

To enable or disable resumable space allocation, you must have the RESUMABLE system privilege.

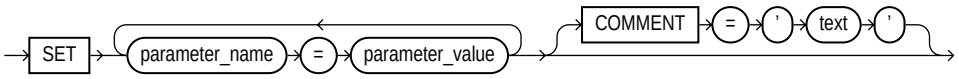
You do not need any privileges to perform the other operations of this statement unless otherwise indicated.

Syntax

alter_session::=



alter_session_set_clause::=



Keywords and Parameters

ADVISE Clause

The ADVISE clause sends advice to a remote database to force a distributed transaction. The advice appears in the ADVICE column of the DBA_2PC_PENDING view on the remote database (the values are 'C' for COMMIT, 'R' for ROLLBACK, and ' ' for NOTHING). If the transaction becomes in doubt, then the administrator of that database can use this advice to decide whether to commit or roll back the transaction.

You can send different advice to different remote databases by issuing multiple ALTER SESSION statements with the ADVISE clause in a single transaction. Each such statement sends advice to the databases referenced in the following statements in the transaction until another such statement is issued.

> **See Also:** "Forcing a Distributed Transaction: Example" on page 10-18

CLOSE DATABASE LINK Clause

Specify CLOSE DATABASE LINK to close the database link *dblink*. When you issue a statement that uses a database link, Oracle creates a session for you on the remote database using that link. The connection remains open until you end your local session or until the number of database links for your session exceeds the value of the initialization parameter OPEN_LINKS. If you want to reduce the network overhead associated with keeping the link open, then use this clause to close the link explicitly if you do not plan to use it again in your session.

> **See Also:** Closing a Database Link: Example on page 10-18

ENABLE | DISABLE COMMIT IN PROCEDURE

Procedures and stored functions written in PL/SQL can issue COMMIT and ROLLBACK statements. If your application would be disrupted by a COMMIT or ROLLBACK statement not issued directly by the application itself, then specify

DISABLE COMMIT IN PROCEDURE clause to prevent procedures and stored functions called during your session from issuing these statements.

You can subsequently allow procedures and stored functions to issue COMMIT and ROLLBACK statements in your session by issuing the ENABLE DISABLE COMMIT IN PROCEDURE.

Some applications (such as SQL*Forms) automatically prohibit COMMIT and ROLLBACK statements in procedures and stored functions. Refer to your application documentation for more information.

PARALLEL DML | DDL | QUERY

The PARALLEL parameter determines whether all subsequent DML, DDL, or query statements in the session will be considered for parallel execution. This clause enables you to override the degree of parallelism of tables during the current session without changing the tables themselves. Uncommitted transactions must either be committed or rolled back prior to executing this clause for DML.

See Also: "Enabling Parallel DML: Example" on page 10-18

ENABLE Clause

Specify ENABLE to execute subsequent statements in the session in parallel. This is the default for DDL and query statements.

- DML: The session's DML statements are executed in parallel mode if a parallel hint or a parallel clause is specified.
- DDL: The session's DDL statements are executed in parallel mode if a parallel clause is specified.
- QUERY: The session's queries are executed in parallel mode if a parallel hint or a parallel clause is specified

Restriction on the ENABLE clause: You cannot specify the optional PARALLEL *integer* with ENABLE.

DISABLE Clause

Specify DISABLE to execute subsequent statements serially. This is the default for DML statements.

- DML: The session's DML statements are executed serially.
- DDL: The session's DDL statements are executed serially.

- QUERY: The session's queries are executed serially.

Restriction on the DISABLE clause: You cannot specify the optional PARALLEL *integer* with DISABLE.

FORCE Clause

FORCE forces parallel execution of subsequent statements in the session. If no parallel clause or hint is specified, then a default degree of parallelism is used. This clause overrides any *parallel_clause* specified in subsequent statements in the session, but is overridden by a parallel hint.

- DML: Provided no parallel DML restrictions are violated, subsequent DML statements in the session are executed with the default degree of parallelism, unless a degree is specified in this clause.
- DDL: Subsequent DDL statements in the session are executed with the default degree of parallelism, unless a degree is specified in this clause. Resulting database objects will have associated with them the prevailing degree of parallelism.

  Using FORCE DDL automatically causes all tables created in this session to be created with a default level of parallelism. The effect is the same as if you had specified the *parallel_clause* (with default degree) with the CREATE TABLE statement.
- QUERY: Subsequent queries are executed with the default degree of parallelism, unless a degree is specified in this clause.

PARALLEL *integer* Specify an integer to explicitly specify a degree of parallelism:

- For FORCE DDL, the degree overrides any parallel clause in subsequent DDL statements.
- For FORCE DML and QUERY, the degree overrides the degree currently stored for the table in the data dictionary.
- A degree specified in a statement through a hint will override the degree being forced.

The following types of DML operations are not parallelized regardless of this clause:

- Operations on clustered tables
- Operations with embedded functions that either write or read database or package states

- Operations on tables with triggers that could fire
- Operations on tables or schema objects containing object types, or LONG or LOB datatypes.

RESUMABLE Clauses

These clauses let you enable and disable resumable space allocation. This feature allows an operation to be suspended in the event of an out-of-space error condition and to resume automatically from the point of interruption when the error condition is fixed.

> **Note:** Resumable space allocation is fully supported for operations on locally managed tablespaces. Some restrictions apply if you are using dictionary-managed tablespaces. For information on these restrictions, please refer to *Oracle9i Database Administrator's Guide*.

ENABLE RESUMABLE

This clause enables resumable space allocation for the session.

TIMEOUT TIMEOUT lets you specify (in seconds) the time during which an operation can remain suspended while waiting for the error condition to be fixed. If the error condition is not fixed within the TIMEOUT period, then Oracle aborts the suspended operation.

NAME NAME lets you specify a user-defined text string to help users identify the statements issued during the session while the session is in resumable mode. Oracle inserts the text string into the USER_RESUMABLE and DBA_RESUMABLE data dictionary views. If you do not specify NAME, then Oracle inserts the default string 'User *username*(*userid*), Session *sessionid*, Instance *instanceid*'.

> **See Also:** *Oracle9i Database Reference* for information on the data dictionary views

DISABLE RESUMABLE

This clause disables resumable space allocation for the session.

alter_session_set_clause

Use the *alter_session_set_clause* to set the parameters that follow (session parameters and initialization parameters that are dynamic in the scope of the

ALTER SESSION statement). You can set values for multiple parameters in the same *alter_session_set_clause*.

COMMENT lets you associate a comment string with this change in the value of the parameter.

Initialization Parameters and ALTER SESSION

All initialization parameters that can be set using an ALTER SYSTEM statement are documented at ALTER SYSTEM on page 10-22. The initialization parameters that are dynamic in the scope of ALTER SESSION are listed in Table 10–1 on page 10-8 with cross-references to their descriptions in ALTER SYSTEM. The only difference in behavior is that when you set these parameters using ALTER SESSION, the value you set persists only for the duration of the current session.

A number of parameters that can be set using ALTER SESSION are not initialization parameters. That is, you can set them only with ALTER SESSION, not in an initialization parameter file. Those session parameters are described *after* Table 10–1.

Caution: Unless otherwise indicated, the parameters described here are initialization parameters, and the descriptions indicate only the general nature of the parameters. Before changing the values of initialization parameters, please refer to their full description in *Oracle9i Database Reference* or *Oracle9i Database Globalization Support Guide*.

Table 10–1 Initialization Parameters You Can Set with ALTER SESSION

Parameter	Comments
CURSOR_SHARING on page 10-46	See also *Oracle9i Database Performance Tuning Guide and Reference* for information on setting this parameter in these and other environments.
DB_BLOCK_CHECKING on page 10-47	The setting made by ALTER SESSION SET DB_BLOCK_CHECKING will be overridden by any subsequent ALTER SYSTEM SET DB_BLOCK_CHECKING statement.
DB_CREATE_FILE_DEST on page 10-49	–
DB_CREATE_ONLINE_LOG_DEST_n on page 10-50	–
DB_FILE_MULTIBLOCK_READ_COUNT on page 10-50	–
FILESYSTEMIO_OPTIONS on page 10-61	–

Table 10–1 Initialization Parameters You Can Set with ALTER SESSION

Parameter	Comments
GLOBAL_NAMES on page 10-62	See "Referring to Objects in Remote Databases" on page 2-118 and *Oracle9i Heterogeneous Connectivity Administrator's Guide* for more information on global name resolution and how Oracle enforces it.
HASH_AREA_SIZE on page 10-63	–
HASH_JOIN_ENABLED on page 10-63	–
LOG_ARCHIVE_DEST_n on page 10-71	–
LOG_ARCHIVE_DEST_STATE_n on page 10-72	–
LOG_ARCHIVE_MIN_SUCCEED_DEST on page 10-74	–
MAX_DUMP_FILE_SIZE on page 10-79	–
Globalization Support (NLS_) Parameters: When you start an instance, Oracle establishes globalization support based on the values of initialization parameters that begin with "NLS". You can query the dynamic performance table V$NLS_PARAMETERS to see the current globalization attributes for your session. For more information about NLS parameters, see *Oracle9i Database Globalization Support Guide.*	
NLS_CALENDAR on page 10-80	–
NLS_COMP on page 10-81	–
NLS_CURRENCY on page 10-81	–
NLS_DATE_FORMAT on page 10-81	See "Date Format Models" on page 2-68 for information on valid date format models.
NLS_DATE_LANGUAGE on page 10-82	–
NLS_DUAL_CURRENCY on page 10-82	See "Number Format Models" on page 2-64 for information on number format elements.
NLS_ISO_CURRENCY on page 10-82	–
NLS_LANGUAGE on page 10-83	–
NLS_LENGTH_SEMANTICS on page 10-83	–
NLS_NCHAR_CONV_EXCP on page 10-83	–
NLS_NUMERIC_CHARACTERS on page 10-84	–

Table 10–1 Initialization Parameters You Can Set with ALTER SESSION

Parameter	Comments
NLS_SORT on page 10-84	–
NLS_TERRITORY on page 10-84	–
NLS_TIMESTAMP_FORMAT on page 10-85	–
NLS_TIMESTAMP_TZ_FORMAT on page 10-85	–
OBJECT_CACHE_MAX_SIZE_PERCENT on page 10-86	–
OBJECT_CACHE_OPTIMAL_SIZE on page 10-86	–
OLAP_PAGE_POOL_SIZE on page 10-86	–
OPTIMIZER_DYNAMIC_SAMPLING on page 10-88	See *Oracle9i Database Performance Tuning Guide and Reference* for information on how to set this parameter.
OPTIMIZER_INDEX_CACHING on page 10-88	–
OPTIMIZER_INDEX_COST_ADJ on page 10-89	–
OPTIMIZER_MAX_PERMUTATIONS on page 10-89	–
OPTIMIZER_MODE on page 10-90	See *Oracle9i Database Concepts* and *Oracle9i Database Performance Tuning Guide and Reference* for information on how to choose a goal for the cost-based approach based on the characteristics of your application.
ORACLE_TRACE_ENABLE on page 10-91	–
PARALLEL_INSTANCE_GROUP on page 10-94	–
PARALLEL_MIN_PERCENT on page 10-95	–
PARTITION_VIEW_ENABLED on page 10-96	For important information on partition views, see "Partition Views" on page 16-40.
PLSQL_COMPILER_FLAGS on page 10-97	–
QUERY_REWRITE_ENABLED on page 10-101	–
QUERY_REWRITE_INTEGRITY on page 10-101	–
REMOTE_DEPENDENCIES_MODE on page 10-102	–
SESSION_CACHED_CURSORS on page 10-107	–

Table 10–1 Initialization Parameters You Can Set with ALTER SESSION

Parameter	Comments
SORT_AREA_RETAINED_SIZE on page 10-111	–
SORT_AREA_SIZE on page 10-112	–
STAR_TRANSFORMATION_ENABLED on page 10-114	–
STATISTICS_LEVEL on page 10-114	–
TIMED_OS_STATISTICS on page 10-115	–
TIMED_STATISTICS on page 10-116	–
TRACE_ENABLED on page 10-116	–
UNDO_SUPPRESS_ERRORS on page 10-119	–
WORKAREA_SIZE_POLICY on page 10-121	–

Session Parameters and ALTER SESSION

The following parameters are session parameters only, not initialization parameters:

CONSTRAINT[S]

Syntax:

```
CONSTRAINT[S] = { IMMEDIATE | DEFERRED | DEFAULT }
```

The `CONSTRAINT[S]` parameter determines when conditions specified by a deferrable constraint are enforced.

- `immediate` indicates that the conditions specified by the deferrable constraint are checked immediately after each DML statement. This setting is equivalent to issuing the `SET CONSTRAINTS ALL IMMEDIATE` statement at the beginning of each transaction in your session.
- `deferred` indicates that the conditions specified by the deferrable constraint are checked when the transaction is committed. This setting is equivalent to issuing the `SET CONSTRAINTS ALL DEFERRED` statement at the beginning of each transaction in your session.
- `default` restores all constraints at the beginning of each transaction to their initial state of `DEFERRED` or `IMMEDIATE`.

CREATE_STORED_OUTLINES

Syntax:

```
CREATE_STORED_OUTLINES = {TRUE | FALSE | 'category_name'}
```

The `CREATE_STORED_OUTLINES` parameter determines whether Oracle should automatically create and store an outline for each query submitted during the session.

- `true` enables automatic outline creation for subsequent queries in the same session. These outlines receive a unique system-generated name and are stored in the `DEFAULT` category. If a particular query already has an outline defined for it in the `DEFAULT` category, then that outline will remain and a new outline will not be created.
- `false` disables automatic outline creation during the session. This is the default.

- *category_name* has the same behavior as TRUE except that any outline created during the session is stored in the *category_name* category.

CURRENT_SCHEMA

Syntax:

```
CURRENT_SCHEMA = schema
```

The CURRENT_SCHEMA parameter changes the current schema of the session to the specified schema. Subsequent unqualified references to schema objects during the session will resolve to objects in the specified schema. The setting persists for the duration of the session or until you issue another ALTER SESSION SET CURRENT_SCHEMA statement.

This setting offers a convenient way to perform operations on objects in a schema other than that of the current user without having to qualify the objects with the schema name. This setting changes the current schema, but it does not change the session user or the current user, nor does it give you any additional system or object privileges for the session.

ERROR_ON_OVERLAP_TIME

Syntax:

```
ERROR_ON_OVERLAP_TIME = {TRUE | FALSE}
```

The ERROR_ON_OVERLAP_TIME determines how Oracle should handle an ambiguous boundary datetime value—that is, a case in which it is not clear whether the datetime is in standard or daylight savings time.

- Specify TRUE to return an error for the ambiguous overlap timestamp.
- Specify FALSE to default the ambiguous overlap timestamp to the standard time. This is the default.

FLAGGER

Syntax:

```
FLAGGER = { ENTRY | INTERMEDIATE | FULL | OFF }
```

The FLAGGER parameter specifies FIPS flagging, which causes an error message to be generated when a SQL statement issued is an extension of ANSI SQL92. FLAGGER is a session parameter only, not an initialization parameter.

In Oracle, there is currently no difference between Entry, Intermediate, or Full level flagging. Once flagging is set in a session, a subsequent ALTER SESSION SET FLAGGER statement will work, but generates the message, ORA-00097. This allows FIPS flagging to be altered without disconnecting the session. OFF turns off flagging.

> **See Also:** Appendix B, "Oracle and Standard SQL", for more information about Oracle compliance with current ANSI SQL standards

INSTANCE

Syntax:

```
INSTANCE = integer
```

The INSTANCE parameter in a Real Application Clusters environment accesses database files as if the session were connected to the instance specified by integer. INSTANCE is a session parameter only, not an initialization parameter. For optimum performance, each instance of Real Application Clusters uses its own private rollback segments, freelist groups, and so on. In a Real Application Clusters environment, you normally connect to a particular instance and access data that is partitioned primarily for your use. If you must connect to another instance, then the data partitioning can be lost. Setting this parameter lets you access an instance as if you were connected to your own instance.

ISOLATION_LEVEL

Syntax:

```
ISOLATION_LEVEL = {SERIALIZABLE | READ COMMITTED}
```

The ISOLATION_LEVEL parameter specifies how transactions containing database modifications are handled. ISOLATION_LEVEL is a session parameter only, not an initialization parameter.

- SERIALIZABLE indicates that transactions in the session use the serializable transaction isolation mode as specified in SQL92. That is, if a serializable transaction attempts to execute a DML statement that updates rows currently being updated by another uncommitted transaction at the start of the serializable transaction, then the DML statement fails. A serializable transaction can see its own updates.
- READ COMMITTED indicates that transactions in the session will use the default Oracle transaction behavior. Thus, if the transaction contains DML that requires

row locks held by another transaction, then the DML statement will wait until the row locks are released.

PLSQL_DEBUG

Syntax:

```
PLSQL_DEBUG = { TRUE | FALSE }
```

The PLSQL_DEBUG parameter sets the default for including or not including debugging information during compile operations. Setting this parameter to TRUE has the same effect as adding the DEBUG keyword to ALTER {FUNCTION | PROCEDURE | PACKAGE} COMPILE statements.

SKIP_UNUSABLE_INDEXES

Syntax:

```
SKIP_UNUSABLE_INDEXES = { TRUE | FALSE }
```

The SKIP_UNUSABLE_INDEXES parameter controls the use and reporting of tables with unusable indexes or index partitions. SKIP_UNUSABLE_INDEXES is a session parameter only, not an initialization parameter.

- TRUE disables error reporting of indexes and index partitions marked UNUSABLE. This setting allows all operations (inserts, deletes, updates, and selects) on tables with unusable indexes or index partitions.

  Note: If an index is used to enforce a UNIQUE constraint on a table, then allowing insert and update operations on the table might violate the constraint. Therefore, this setting does not disable error reporting for unusable indexes that are unique.

- FALSE enables error reporting of indexes marked UNUSABLE. This setting does not allow inserts, deletes, and updates on tables with unusable indexes or index partitions. This is the default.

SQL_TRACE

Syntax:

```
INSTANCE = integer
```

SQL_TRACE is an initialization parameter. However, when you change its value with an ALTER SESSION statement, the results are not reflected in the V$PARAMETER view. Therefore, in this context it is considered a session parameter only.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for more information on the SQL trace facility, including how to format and interpret its output

TIME_ZONE

Syntax:

```
TIME_ZONE =  '[+ | -] hh:mm'
             | LOCAL
             | DBTIMEZONE
             | 'time_zone_region'
```

The TIME_ZONE parameter specifies the default local time zone displacement for the current SQL session. TIME_ZONE is a session parameter only, not an initialization parameter.

- Specify a format mask (`'[+|-]hh:mm'`) indicating the hours and minutes before or after UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The valid range for *hh:mm* is -12:00 to +14:00.
- Specify LOCAL to set the default local time zone displacement of the current SQL session to the original default local time zone displacement that was established when the current SQL session was started.
- Specify DBTIMEZONE to set the current session time zone to match the value set for the database time zone. If you specify this setting, then the DBTIMEZONE function will return the database time zone as a UTC offset or a time zone region, depending on how the database time zone has been set.
- Specify a valid *time_zone_region*. To see a listing of valid region names, query the TZNAME column of the V$TIMEZONE_NAMES dynamic performance view. If you specify this setting, then the SESSIONTIMEZONE function will return the region name.

  > **Note:** You can also set the default client session time zone using the ORA_SDTZ environment variable. Please refer to *Oracle9i Database Globalization Support Guide* for more information on this variable.

USE_PRIVATE_OUTLINES

Syntax:

```
USE_PRIVATE_OUTLINES = { TRUE | FALSE | category_name }
```

The USE_PRIVATE_OUTLINES parameter lets you control the use of private outlines. When this parameter is enabled and an outlined SQL statement is issued, the optimizer retrieves the outline from the session private area rather than the public area used when USE_STORED_OUTLINES is enabled. If no outline exists in the session private area, then the optimizer will not use an outline to compile the statement. USE_PRIVATE_OUTLINES is not an initialization parameter.

- TRUE causes the optimizer to use private outlines stored in the DEFAULT category when compiling requests.
- FALSE specifies that the optimizer should not use stored private outlines. This is the default. If USE_STORED_OUTLINES is enabled, then the optimizer will use stored public outlines.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

Restriction on USE_PRIVATE_OUTLINES: You cannot enable this parameter if USE_STORED_OUTLINES is enabled.

USE_STORED_OUTLINES

Syntax:

```
USE_STORED_OUTLINES = { TRUE | FALSE | category_name }
```

The USE_STORED_OUTLINES parameter determines whether the optimizer will use stored public outlines to generate execution plans. USE_STORED_OUTLINES is not an initialization parameter.

- TRUE causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- FALSE specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

Restriction on USED_STORED_OUTLINES: You cannot enable this parameter if USE_PRIVATE_OUTLINES is enabled.

Examples

Enabling Parallel DML: Example Issue the following statement to enable parallel DML mode for the current session:

```
ALTER SESSION ENABLE PARALLEL DML;
```

Forcing a Distributed Transaction: Example The following transaction inserts an employee record into the `employees` table on the database identified by the database link `site1` and deletes an employee record from the `employees` table on the database identified by `site2`:

```
ALTER SESSION
   ADVISE COMMIT;

INSERT INTO employees@site1
   VALUES (8002, 'Juan', 'Fernandez', 'juanf@hr.com', NULL,
   TO_DATE('04-OCT-1992', 'DD-MON-YYYY'), 'SA_CLERK', 3000,
   NULL, 121, 20);

ALTER SESSION
   ADVISE ROLLBACK;

DELETE FROM employees@site2
   WHERE employee_id = 8002;

COMMIT;
```

This transaction has two `ALTER SESSION` statements with the `ADVISE` clause. If the transaction becomes in doubt, then `site1` is sent the advice '`COMMIT`' by virtue of the first `ALTER SESSION` statement and `site2` is sent the advice '`ROLLBACK`' by virtue of the second.

Closing a Database Link: Example This statement updates the employee table on the `hq` database using a database link, commits the transaction, and explicitly closes the database link:

```
UPDATE employees@hq
   SET salary = salary + 200
   WHERE employee_id = 162;

COMMIT;

ALTER SESSION
   CLOSE DATABASE LINK hq;
```

Changing the Date Format Dynamically: Example The following statement dynamically changes the default date format for your session to 'YYYY MM DD-HH24:MI:SS':

```
ALTER SESSION
   SET NLS_DATE_FORMAT = 'YYYY MM DD HH24:MI:SS';
```

Oracle uses the new default date format:

```
SELECT TO_CHAR(SYSDATE) Today
   FROM DUAL;

TODAY
-------------------
2001 04 12 12:30:38
```

Changing the Date Language Dynamically: Example The following statement changes the language for date format elements to French:

```
ALTER SESSION
   SET NLS_DATE_LANGUAGE = French;

SELECT TO_CHAR(SYSDATE, 'Day DD Month YYYY') Today
   FROM DUAL;

TODAY
---------------------------
Jeudi    12 Avril     2001
```

Changing the ISO Currency: Example The following statement dynamically changes the ISO currency symbol to the ISO currency symbol for the territory America:

```
ALTER SESSION
   SET NLS_ISO_CURRENCY = America;

SELECT TO_CHAR( SUM(salary), 'C999G999D99') Total
   FROM employees;

TOTAL
------------------
     USD694,900.00
```

Changing the Decimal Character and Group Separator: Example The following statement dynamically changes the decimal character to comma (,) and the group separator to period (.):

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ',.' ;
```

Oracle returns these new characters when you use their number format elements:

```
ALTER SESSION SET NLS_CURRENCY = 'FF';

SELECT TO_CHAR( SUM(salary), 'L999G999D99') Total FROM employees;

TOTAL
---------------------
         FF694.900,00
```

Changing the NLS Currency: Example The following statement dynamically changes the local currency symbol to 'DM':

```
ALTER SESSION
   SET NLS_CURRENCY = 'DM';

SELECT TO_CHAR( SUM(salary), 'L999G999D99') Total
   FROM employees;

TOTAL
---------------------
         DM694.900,00
```

Changing the NLS Language: Example The following statement dynamically changes to French the language in which error messages are displayed:

```
ALTER SESSION
   SET NLS_LANGUAGE = FRENCH;

Session modifiee.

SELECT * FROM DMP;

ORA-00942: Table ou vue inexistante
```

Changing the Linguistic Sort Sequence: Example The following statement dynamically changes the linguistic sort sequence to Spanish:

```
ALTER SESSION
   SET NLS_SORT = XSpanish;
```

Oracle sorts character values based on their position in the Spanish linguistic sort sequence.

Enabling SQL Trace: Example To enable the SQL trace facility for your session, issue the following statement:

```
ALTER SESSION
   SET SQL_TRACE = TRUE;
```

Enabling Query Rewrite: Example This statement enables query rewrite in the current session for all materialized views that have not been explicitly disabled:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

ALTER SYSTEM

Purpose

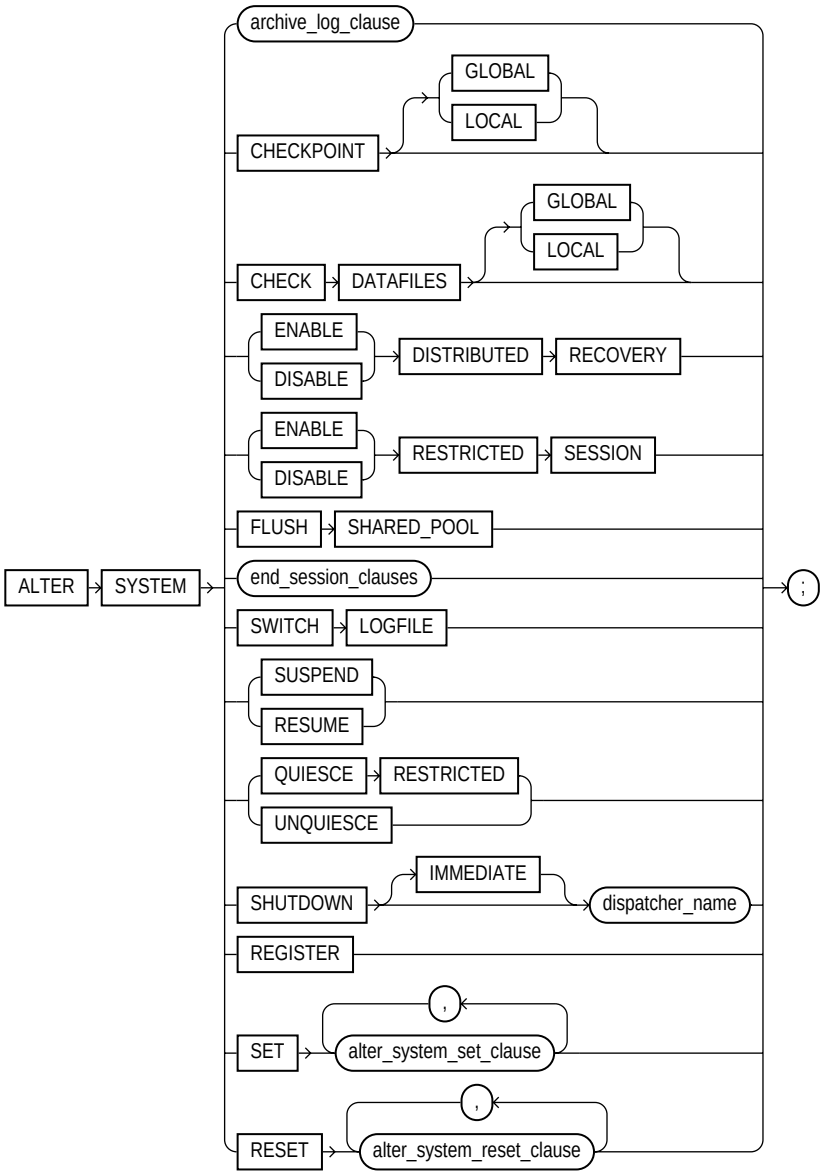
Use the ALTER SYSTEM statement to dynamically alter your Oracle instance. The settings stay in effect as long as the database is mounted.

Prerequisites

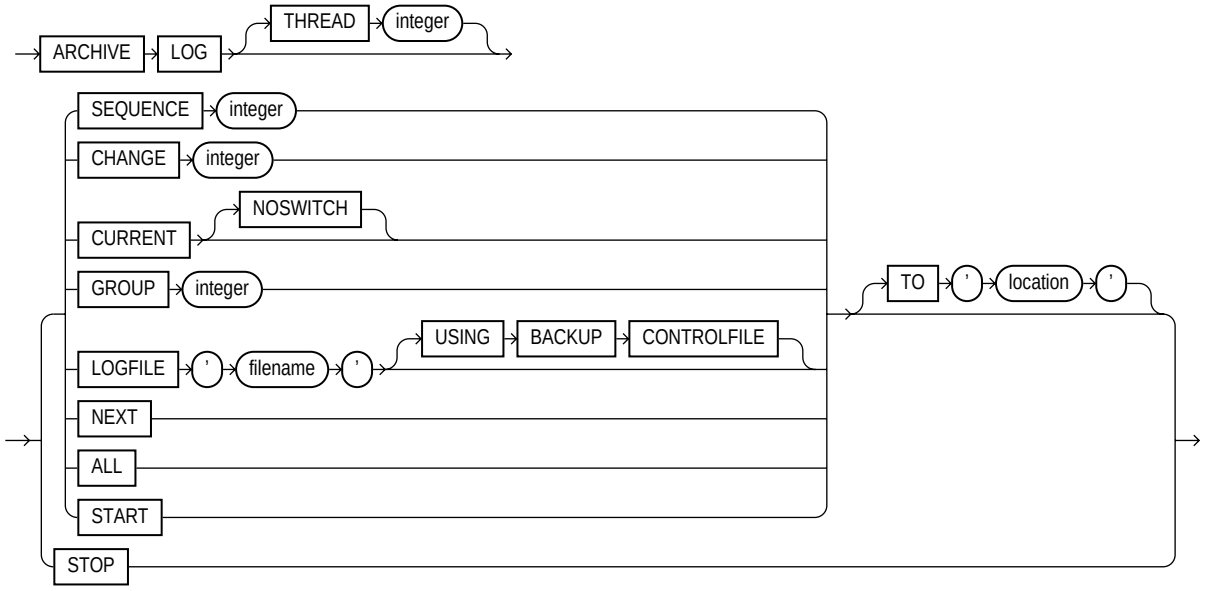
You must have ALTER SYSTEM system privilege.

Syntax

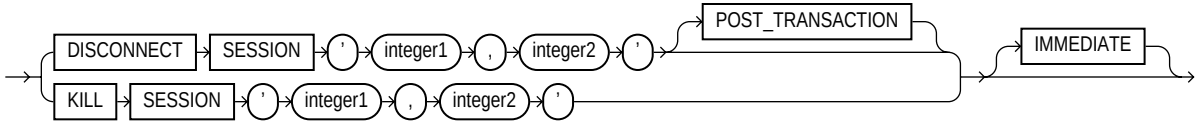
alter_system::=



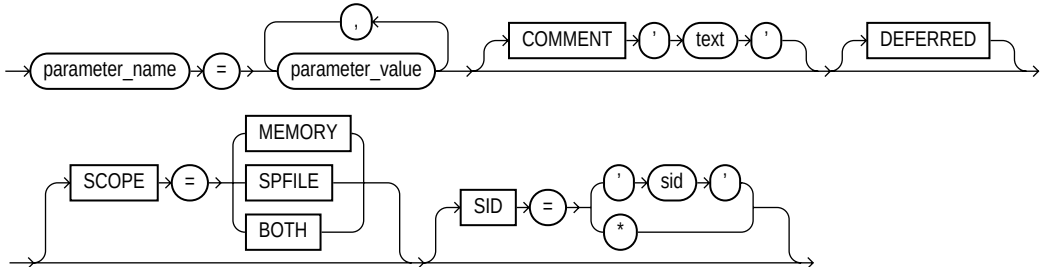
archive_log_clause::=



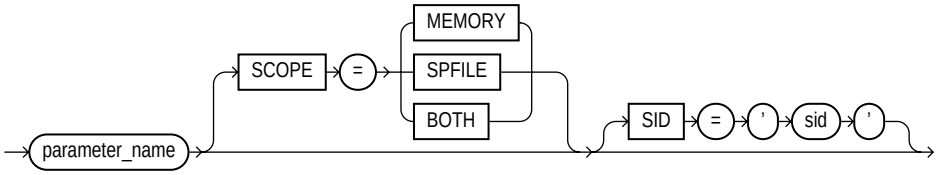
end_session_clauses::=



alter_system_set_clause::=



alter_system_reset_clause::=



Keywords and Parameters

archive_log_clause

The *archive_log_clause* manually archives redo log files or enables or disables automatic archiving. To use this clause, your instance must have the database mounted. The database can be either open or closed unless otherwise noted.

THREAD Clause

Specify THREAD to indicate the thread containing the redo log file group to be archived.

Restriction on the THREAD clause: Set this parameter only if you are using Oracle with Real Application Clusters.

> **See Also:** "Archiving Redo Logs Manually: Examples" on page 10-121

SEQUENCE Clause

Specify SEQUENCE to manually archive the online redo log file group identified by the log sequence number *integer* in the specified thread. If you omit the THREAD parameter, then Oracle archives the specified group from the thread assigned to your instance.

CHANGE Clause

Specify CHANGE to manually archive the online redo log file group containing the redo log entry with the system change number (SCN) specified by *integer* in the specified thread. If the SCN is in the current redo log file group, then Oracle performs a log switch. If you omit the THREAD parameter, then Oracle archives the groups containing this SCN from all enabled threads.

You can use this clause only when your instance has the database open.

CURRENT Clause

Specify CURRENT to manually archive the current redo log file group of the specified thread, forcing a log switch. If you omit the THREAD parameter, then Oracle archives all redo log file groups from all enabled threads, including logs previous to current logs. You can specify CURRENT only when the database is open.

NOSWITCH Specify NOSWITCH if you want to manually archive the current redo log file group without forcing a log switch. This setting is used primarily with standby databases to prevent data divergence when the primary database shuts down. Divergence implies the possibility of data loss in case of primary database failure.

You can use the NOSWITCH clause only when your instance has the database mounted but not open. If the database is open, then this operation closes the database automatically. You must then manually shut down the database before you can reopen it.

GROUP Clause

Specify GROUP to manually archive the online redo log file group with the GROUP value specified by *integer*. You can determine the GROUP value for a redo log file group by querying the data dictionary view DBA_LOG_FILES. If you specify both the THREAD and GROUP parameters, then the specified redo log file group must be in the specified thread.

LOGFILE Clause

Specify LOGFILE to manually archive the online redo log file group containing the redo log file member identified by '*filename*'. If you specify both the THREAD and LOGFILE parameters, then the specified redo log file group must be in the specified thread.

If the database was mounted with a backup controlfile, then specify USING BACKUP CONTROLFILE to permit archiving of all online logfiles, including the current logfile.

Restriction on the LOGFILE clause: You must archive redo log file groups in the order in which they are filled. If you specify a redo log file group for archiving with the LOGFILE parameter, and earlier redo log file groups are not yet archived, then Oracle returns an error.

NEXT Clause

Specify NEXT to manually archive the next online redo log file group from the specified thread that is full but has not yet been archived. If you omit the THREAD parameter, then Oracle archives the earliest unarchived redo log file group from any enabled thread.

ALL Clause

Specify ALL to manually archive all online redo log file groups from the specified thread that are full but have not been archived. If you omit the THREAD parameter, then Oracle archives all full unarchived redo log file groups from all enabled threads.

START Clause

Specify START to enable automatic archiving of redo log file groups.

Restriction on the START clause: You can enable automatic archiving only for the thread assigned to your instance.

TO *location* Clause

Specify TO '*location*' to indicate the primary location to which the redo log file groups are archived. The value of this parameter must be a fully specified file location following the conventions of your operating system. If you omit this parameter, then Oracle archives the redo log file group to the location specified by the initialization parameters LOG_ARCHIVE_DEST or LOG_ARCHIVE_DEST_*n*.

STOP Clause

Specify STOP to disable automatic archiving of redo log file groups. You can disable automatic archiving only for the thread assigned to your instance.

CHECKPOINT Clause

Specify CHECKPOINT to explicitly force Oracle to perform a checkpoint, ensuring that all changes made by committed transactions are written to datafiles on disk. You can specify this clause only when your instance has the database open. Oracle does not return control to you until the checkpoint is complete.

GLOBAL In a Real Application Clusters environment, this setting causes Oracle to perform a checkpoint for all instances that have opened the database. This is the default.

LOCAL In a Real Application Clusters environment, this setting causes Oracle to perform a checkpoint only for the thread of redo log file groups for the instance from which you issue the statement.

See Also: "Forcing a Checkpoint: Example" on page 10-122

CHECK DATAFILES Clause

In a distributed database system, such as a Real Application Clusters environment, this clause updates an instance's SGA from the database control file to reflect information on all online datafiles.

- Specify GLOBAL to perform this synchronization for all instances that have opened the database. This is the default.
- Specify LOCAL to perform this synchronization only for the local instance.

Your instance should have the database open.

end_session_clauses

The *end_session_clauses* give you several ways to end the current session.

DISCONNECT SESSION Clause

Use the DISCONNECT SESSION clause to disconnect the current session by destroying the dedicated server process (or virtual circuit if the connection was made by way of a Shared Server). To use this clause, your instance must have the database open. You must identify the session with both of the following values from the V$SESSION view:

- For *integer1*, specify the value of the SID column.
- For *integer2*, specify the value of the SERIAL# column.

If system parameters are appropriately configured, then application failover will take effect.

- The POST_TRANSACTION setting allows ongoing transactions to complete before the session is disconnected. If the session has no ongoing transactions, then this clause has the same effect described for as KILL SESSION.
- The IMMEDIATE setting disconnects the session and recovers the entire session state immediately, without waiting for ongoing transactions to complete.

- If you also specify POST_TRANSACTION and the session has ongoing transactions, then the IMMEDIATE keyword is ignored.
- If you do not specify POST_TRANSACTION, or you specify POST_TRANSACTION but the session has no ongoing transactions, then this clause has the same effect as described for KILL SESSION IMMEDIATE.

See Also: "Disconnecting a Session: Example" on page 10-125

KILL SESSION Clause

The KILL SESSION clause lets you mark a session as terminated, roll back ongoing transactions, release all session locks, and partially recover session resources. To use this clause, your instance must have the database open, and your session and the session to be killed must be on the same instance. You must identify the session with both of the following values from the V$SESSION view:

- For *integer1*, specify the value of the SID column.
- For *integer2*, specify the value of the SERIAL# column.

If the session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, then Oracle waits for this activity to complete, marks the session as terminated, and then returns control to you. If the waiting lasts a minute, then Oracle marks the session to be killed and returns control to you with a message that the session is marked to be killed. The PMON background process then marks the session as terminated when the activity is complete.

Whether or not the session has an ongoing transaction, Oracle does not recover the entire session state until the session user issues a request to the session and receives a message that the session has been killed.

See Also: "Killing a Session: Example" on page 10-124

IMMEDIATE Specify IMMEDIATE to instruct Oracle to roll back ongoing transactions, release all session locks, recover the entire session state, and return control to you immediately.

DISTRIBUTED RECOVERY Clause

The DISTRIBUTED RECOVERY clause lets you enable or disable distributed recovery. To use this clause, your instance must have the database open.

ENABLE Specify ENABLE to enable distributed recovery. In a single-process environment, you must use this clause to initiate distributed recovery.

You may need to issue the ENABLE DISTRIBUTED RECOVERY statement more than once to recover an in-doubt transaction if the remote node involved in the transaction is not accessible. In-doubt transactions appear in the data dictionary view DBA_2PC_PENDING.

> **See Also:** "Enabling Distributed Recovery: Example" on page 10-124

DISABLE Specify DISABLE to disable distributed recovery.

RESTRICTED SESSION Clause

The RESTRICTED SESSION clause lets you restrict logon to Oracle.

You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

ENABLE Specify ENABLE to allows only users with RESTRICTED SESSION system privilege to log on to Oracle. Existing sessions are not terminated.

DISABLE Specify DISABLE to reverse the effect of the ENABLE RESTRICTED SESSION clause, allowing all users with CREATE SESSION system privilege to log on to Oracle. This is the default.

> **See Also:** "Restricting Session Logons: Example" on page 10-122

FLUSH SHARED_POOL Clause

The FLUSH SHARED POOL clause lets you clear all data from the shared pool in the system global area (SGA). The shared pool stores

- Cached data dictionary information and
- Shared SQL and PL/SQL areas for SQL statements, stored procedures, function, packages, and triggers.

This statement does not clear shared SQL and PL/SQL areas for items that are currently being executed. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

> **See Also:** "Clearing the Shared Pool: Example" on page 10-122

SWITCH LOGFILE Clause

The SWITCH LOGFILE clause lets you explicitly force Oracle to begin writing to a new redo log file group, regardless of whether the files in the current redo log file group are full. When you force a log switch, Oracle begins to perform a checkpoint but returns control to you immediately rather than when the checkpoint is complete. To use this clause, your instance must have the database open.

See Also: "Forcing a Log Switch: Example" on page 10-124

SUSPEND | RESUME

The SUSPEND clause lets you suspend all I/O (datafile, control file, and file header) as well as queries, in all instances, enabling you to make copies of the database without having to handle ongoing transactions.

Restrictions on SUSPEND and RESUME:

- Do not use this clause unless you have put the database tablespaces in hot backup mode.
- If you start a new instance while the system is suspended, then that new instance will not be suspended.

The RESUME clause lets you make the database available once again for queries and I/O.

QUIESCE RESTRICTED | UNQUIESCE

Use the QUIESCE RESTRICTED and UNQUIESCE clauses to put the database in and take it out of the **quiesced state**. This state enables database administrators to perform administrative operations that cannot be safely performed in the presence of concurrent transactions, queries, or PL/SQL operations.

Note: The QUIESCE RESTRICTED clause is valid only if the Database Resource Manager feature is installed and only if the Resource Manager has been on continuously since database startup in any instances that have opened the database.

If multiple QUIESCE RESTRICTED or UNQUIESCE statements issue at the same time from different sessions or instances, then all but one will receive an error.

QUIESCE RESTRICTED

Specify QUIESCE RESTRICTED to put the database in the quiesced state. For all instances with the database open, this clause has the following effect:

- Oracle instructs the Database Resource Manager in all instances to prevent all inactive sessions (other than SYS and SYSTEM) from becoming active. No user other than SYS and SYSTEM can start a new transaction, a new query, a new fetch, or a new PL/SQL operation.
- Oracle waits for all existing transactions in all instances that were initiated by a user other than SYS or SYSTEM to finish (either commit or abort). Oracle also waits for all running queries, fetches, and PL/SQL procedures in all instances that were initiated by users other than SYS or SYSTEM and that are not inside transactions to finish. If a query is carried out by multiple successive OCI fetches, then Oracle does not wait for all fetches to finish. It waits for the current fetch to finish and then blocks the next fetch. Oracle also waits for all sessions (other than those of SYS or SYSTEM) that hold any shared resources (such as enqueues) to release those resources. After all these operations finish, Oracle places the database into quiesced state and finishes executing the QUIESCE RESTRICTED statement.
- If an instance is running in shared server mode, then Oracle instructs the Database Resource Manager to block logins (other than SYS or SYSTEM) on that instance. If an instance is running in non-shared-server mode, then Oracle does not impose any restrictions on user logins in that instance.

During the quiesced state, you cannot change the Resource Manager plan in any instance.

UNQUIESCE

Specify UNQUIESCE to take the database out of quiesced state. Doing so permits transactions, queries, fetches, and PL/SQL procedures that were initiated by users other than SYS or SYSTEM to be undertaken once again. The UNQUIESCE statement does not have to originate in the same session that issued the QUIESCE RESTRICTED statement.

SHUTDOWN Clause

The SHUTDOWN clause is relevant only if your system is using Oracle's shared server architecture. It shuts down a dispatcher identified by *dispatcher_name*. The *dispatcher_name* must be a string of the form 'D*xxx*', where xxx indicates the number of the dispatcher. For a listing of dispatcher names, query the NAME column of the V$DISPATCHER dynamic performance view.

- If you specify IMMEDIATE, then the dispatcher stops accepting new connections immediately and Oracle terminates all existing connections through that dispatcher. After all sessions are cleaned up, the dispatcher process literally shuts down.
- If you do not specify IMMEDIATE, then the dispatcher stops accepting new connections immediately but waits for all its users to disconnect and for all its database links to terminate. Then it literally shuts down.

REGISTER Clause

Specify REGISTER to instruct the PMON background process to register the instance with the listeners immediately. If you do not specify this clause, then registration of the instance does not occur until the next time PMON executes the discovery routine. As a result, clients may not be able to access the services for as long as 60 seconds after the listener is started.

> **See Also:** *Oracle9i Database Concepts* and *Oracle9i Net Services Administrator's Guide* for information on the PMON background process and listeners

alter_system_set_clause

The *alter_system_set_clause* lets you set or reset the value of any initialization parameter. The parameters are described in "Initialization Parameters and ALTER SYSTEM" on page 10-36.

The ability to change initialization parameter values depends on whether you have started up the database with a traditional parameter file (pfile) or with a server parameter file (spfile). To determine whether you can change the value of a particular parameter, query the ISSYS_MODIFIABLE column of the V$PARAMETER dynamic performance view.

When setting the parameter's value, you can specify additional settings as follows:

COMMENT

The COMMENT clause lets you associate a comment string with this change in the value of the parameter. If you also specify SPFILE, then this comment will appear in the parameter file to indicate the most recent change made to this parameter.

DEFERRED

The DEFERRED keyword sets or modifies the value of the parameter for future sessions that connect to the database. Current sessions retain the old value.

You must specify DEFERRED if the value of the ISSYS_MODIFIABLE column of V$PARAMETER for this parameter is DEFERRED. If the value of that column is IMMEDIATE, then the DEFERRED keyword in this clause is optional. If the value of that column is FALSE, then you cannot specify DEFERRED in this ALTER SYSTEM statement.

> **See Also:** *Oracle9i Database Reference* for information on the V$PARAMETER dynamic performance view

SCOPE

The SCOPE clause lets you specify when the change takes effect. Scope depends on whether you are started up the database using a parameter file (pfile) or server parameter file (spfile).

MEMORY MEMORY indicates that the change is made in memory, takes effect immediately, and persists until the database is shut down. If you started up the database using a parameter file (pfile), then this is the only scope you can specify.

SPFILE SPFILE indicates that the change is made in the server parameter file. The new setting takes effect when the database is next shut down and started up again. You must specify SPFILE when changing the value of a static parameter.

BOTH BOTH indicates that the change is made in memory and in the server parameter file. The new setting takes effect immediately and persists after the database is shut down and started up again.

If a server parameter file was used to start up the database, then BOTH is the default. If a parameter file was used to start up the database, then MEMORY is the default, as well as the only scope you can specify.

SID

The SID clause is relevant only in a Real Application Clusters environment. This clause lets you specify the SID of the instance where the value will take effect.

- Specify SID = '*' if you want Oracle to change the value of the parameter for all instances.
- Specify SID = *'sid'* if you want Oracle to change the value of the parameter only for the instance *sid*. This setting takes precedence over previous and subsequent ALTER SYSTEM SET statements that specify SID = '*'.

If you do not specify this clause:

- If the instance was started up with a pfile (client-side initialization parameter file), then Oracle assumes the SID of the current instance.
- If the instance was started up with an spfile (server parameter file), then Oracle assumes SID = '*'.

If you specify an instance other than the current instance, then Oracle sends a message to that instance to change the parameter value in the memory of that instance.

> **See Also:** *Oracle9i Database Reference* for information about the V$PARAMETER view

alter_system_reset_clause

The *alter_system_reset_clause* is for use in a Real Application Clusters environment. It gives you separate control for an individual instance over parameters that may have been set for all instances in a server parameter file. The SCOPE clause has the same behavior as described for the *alter_system_set_clause*.

SID Specify the SID clause to remove a previously specified setting of this parameter for your instance (that is, a previous ALTER SYSTEM SET ... SID = *'sid'* statement). Your instance will assume the value of the parameter as specified in a previous or subsequent ALTER SYSTEM SET ... SID = '*' statement.

> **See Also:** *Oracle9i Real Application Clusters Deployment and Performance* for information on setting parameter values for an individual instance in a Real Application Clusters environment

Initialization Parameters and ALTER SYSTEM

This section contains an alphabetical listing of all initialization parameters with brief descriptions only. For a complete description of these parameters, please refer to their full description in *Oracle9i Database Reference*. *Oracle9i Database Reference*

ACTIVE_INSTANCE_COUNT

Parameter type	Integer
Default value	There is no default value.
Parameter class	Static
Range of values	1 or >= the number of instances in the cluster. (Values other than 1 have no effect on the active or standby status of any instances.)
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have the same value.

Note: This parameter functions only in a cluster with only two instances.

`ACTIVE_INSTANCE_COUNT` enables you to designate one instance in a two-instance cluster as the primary instance and the other instance as the secondary instance. This parameter has no functionality in a cluster with more than two instances.

AQ_TM_PROCESSES

Parameter type	Integer
Default value	0
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	0 to 10

`AQ_TM_PROCESSES` enables time monitoring of queue messages. The times can be used in messages that specify delay and expiration properties. Values from 1 to 10 specify the number of queue monitor processes created to monitor the messages. If

AQ_TM_PROCESSES is not specified or is set to 0, then the queue monitor is not created.

ARCHIVE_LAG_TARGET

Parameter type	Integer
Default value	0 (disabled)
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 or any integer in [60, 7200]
Real Application Clusters	Multiple instances should use the same value.

ARCHIVE_LAG_TARGET limits the amount of data that can be lost and effectively increases the availability of the standby database by forcing a log switch after a user-specified time period elapses.

AUDIT_FILE_DEST

Parameter type	String
Syntax	AUDIT_FILE_DEST = *'directory'*
Default value	*ORACLE_HOME*/rdbms/audit
Parameter class	Static

AUDIT_FILE_DEST specifies the directory where Oracle stores auditing files.

AUDIT_SYS_OPERATIONS

Parameter type	Boolean
Default value	false
Parameter class	Static
Range of values	true | false

AUDIT_SYS_OPERATIONS enables or disables the auditing of operations issued by user SYS, and users connecting with SYSDBA or SYSOPER privileges. The audit records are written to the operating system's audit trail.

AUDIT_TRAIL

Parameter type	String
Syntax	`AUDIT_TRAIL = {NONE | FALSE | DB | TRUE | OS}`
Default value	There is no default value.
Parameter class	Static

`AUDIT_TRAIL` enables or disables the automatic writing of rows to the audit trail.

BACKGROUND_CORE_DUMP

Parameter type	String
Syntax	`BACKGROUND_CORE_DUMP = {partial | full}`
Default value	`partial`
Parameter class	Static

`BACKGROUND_CORE_DUMP` specifies whether Oracle includes the SGA in the core file for Oracle background processes.

BACKGROUND_DUMP_DEST

Parameter type	String
Syntax	`BACKGROUND_DUMP_DEST = {`*`pathname`* `|` *`directory`*`}`
Default value	Operating system-dependent
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	Any valid local path, directory, or disk

`BACKGROUND_DUMP_DEST` specifies the pathname (directory or disc) where debugging trace files for the background processes (LGWR, DBW*n*, and so on) are written during Oracle operations.

BACKUP_TAPE_IO_SLAVES

Parameter type	Boolean
Default value	`false`

Parameter class Dynamic: ALTER SYSTEM ... DEFERRED

Range of values true | false

BACKUP_TAPE_IO_SLAVES specifies whether I/O server processes (also called **slaves**) are used by the Recovery Manager to back up, copy, or restore data to tape. When the value is set to true, Oracle uses an I/O server process to write to or read from a tape device. When the value is set to false (the default), Oracle does not use I/O server process for backups. Instead, the shadow process engaged in the backup accesses the tape device.

BITMAP_MERGE_AREA_SIZE

Parameter type Integer

Default value 1048576 (1 MB)

Parameter class Static

Range of values Operating system-dependent

> **Note:** Oracle does not recommend using the BITMAP_MERGE_AREA_SIZE parameter unless the instance is configured with the shared server option. Oracle recommends that you enable automatic sizing of SQL working areas by setting PGA_AGGREGATE_TARGET instead. BITMAP_MERGE_AREA_SIZE is retained for backward compatibility.

BITMAP_MERGE_AREA_SIZE is relevant only for systems containing bitmap indexes. It specifies the amount of memory Oracle uses to merge bitmaps retrieved from a range scan of the index. The default value is 1 MB. A larger value usually improves performance, because the bitmap segments must be sorted before being merged into a single bitmap.

BLANK_TRIMMING

Parameter type Boolean

Default value false

Parameter class Static

Range of values `true | false`

`BLANK_TRIMMING` specifies the data assignment semantics of character datatypes.

BUFFER_POOL_KEEP

Parameter type String

Syntax `BUFFER_POOL_KEEP = {`*`integer`* `|`

`(BUFFERS:`*`integer`*`, LRU_LATCHES:`*`integer`*`)}`

where *`integer`* is the number of buffers and, optionally, the number of LRU latches.

Default value There is no default value.

Parameter class Static

Note: This parameter is deprecated in favor of the `DB_KEEP_CACHE_SIZE` parameter. Oracle recommends that you use `DB_KEEP_CACHE_SIZE` instead. Also, `BUFFER_POOL_KEEP` cannot be combined with the new dynamic `DB_KEEP_CACHE_SIZE` parameter; combining these parameters in the same parameter file will produce an error. `BUFFER_POOL_KEEP` is retained for backward compatibility only.

`BUFFER_POOL_KEEP` lets you save objects in the buffer cache by setting aside a portion of the total number of buffers (the value of the `DB_BLOCK_BUFFERS` parameter) as a `KEEP` buffer pool. You can also allocate to the `KEEP` buffer pool a specified portion of the total number of LRU latches.

BUFFER_POOL_RECYCLE

Parameter type String

Syntax `BUFFER_POOL_RECYCLE = {`*`integer`* `|`

`(BUFFERS:`*`integer`*`, LRU_LATCHES:`*`integer`*`)}`

where *`integer`* is the number of buffers and, optionally, the number of LRU latches.

Default value There is no default value.

Parameter class Static

> **Note:** This parameter is deprecated in favor of the DB_RECYCLE_CACHE_SIZE parameter. Oracle recommends that you use DB_RECYCLE_CACHE_SIZE instead. Also, BUFFER_POOL_RECYCLE cannot be combined with the new dynamic DB_RECYCLE_CACHE_SIZE parameter; combining these parameters in the same parameter file will produce an error. BUFFER_POOL_RECYCLE is retained for backward compatibility only.

BUFFER_POOL_RECYCLE lets you limit the size of objects in the buffer cache by setting aside a portion of the total number of buffers (the value of the DB_BLOCK_BUFFERS parameter) as a RECYCLE buffer pool. You can also allocate to the RECYCLE buffer pool a specified portion of the total number of LRU latches.

CIRCUITS

Parameter type Integer

Default value Derived:

- If you are using shared server architecture, then the value of SESSIONS
- If you are not using the shared server architecture, then the value is 0

Parameter class Static

CIRCUITS specifies the total number of virtual circuits that are available for inbound and outbound network sessions. It is one of several parameters that contribute to the total SGA requirements of an instance.

CLUSTER_DATABASE

Parameter type Boolean

Default value false

Parameter class Static

Range of values	`true | false`
Real Application Clusters	Multiple instances must have the same value.

CLUSTER_DATABASE is an Oracle9*i* Real Application Clusters parameter that specifies whether or not Oracle9*i* Real Application Clusters is enabled.

CLUSTER_DATABASE_INSTANCES

Parameter type	Integer
Default value	`1`
Parameter class	Static
Range of values	Any nonzero value

CLUSTER_DATABASE_INSTANCES is an Oracle9*i* Real Application Clusters parameter that specifies the number of instances that are configured as part of your cluster database. You must set this parameter for every instance. Normally you should set this parameter to the number of instances in your Oracle9*i* Real Application Clusters environment. A proper setting for this parameter can improve memory use.

CLUSTER_INTERCONNECTS

Parameter type	String
Syntax	CLUSTER_INTERCONNECTS = *ifn* [: *ifn* ...]
Default value	There is no default value.
Parameter class	Static
Range of values	One or more IP addresses, separated by colons

CLUSTER_INTERCONNECTS provides Oracle with information about additional cluster interconnects available for use in Oracle9*i* Real Application Clusters environments.

COMMIT_POINT_STRENGTH

Parameter type	Integer

Default value	1
Parameter class	Static
Range of values	0 to 255

COMMIT_POINT_STRENGTH is relevant only in distributed database systems. It specifies a value that determines the **commit point site** in a distributed transaction. The node in the transaction with the highest value for COMMIT_POINT_STRENGTH will be the commit point site.

COMPATIBLE

Parameter type	String
Syntax	COMPATIBLE = *release_number*
Default value	8.1.0
Parameter class	Static
Range of values	Default release to current release
Real Application Clusters	Multiple instances must have the same value.

COMPATIBLE allows you to use a new release, while at the same time guaranteeing backward compatibility with an earlier release. This is helpful if it becomes necessary to revert to the earlier release.

CONTROL_FILE_RECORD_KEEP_TIME

Parameter type	Integer
Default value	7 (days)
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to 365 (days)

CONTROL_FILE_RECORD_KEEP_TIME specifies the minimum number of days before a reusable record in the control file can be reused. In the event a new record needs to be added to a reusable section and the oldest record has not aged enough, the record section expands. If this parameter is set to 0, then reusable sections never expand, and records are reused as needed.

CONTROL_FILES

Parameter type	String
Syntax	CONTROL_FILES = *filename* [, *filename* [...]] **Note:** The control file name can be an OMF (Oracle Managed Files) name. This occurs when the control file is re-created using the CREATE CONTROLFILE REUSE statement.
Default value	Operating system-dependent
Parameter class	Static
Range of values	1 to 8 filenames
Real Application Clusters	Multiple instances must have the same value.

Every database has a **control file**, which contains entries that describe the structure of the database (such as its name, the timestamp of its creation, and the names and locations of its datafiles and redo files). CONTROL_FILES specifies one or more names of control files, separated by commas.

CORE_DUMP_DEST

Parameter type	String
Syntax	CORE_DUMP_DEST = *directory*
Default value	*ORACLE_HOME*/DBS
Parameter class	Dynamic: ALTER SYSTEM

CORE_DUMP_DEST is primarily a UNIX parameter and may not be supported on your platform. It specifies the directory where Oracle dumps core files.

CPU_COUNT

Parameter type	Integer
Default value	Set automatically by Oracle
Parameter class	Static
Range of values	0 to unlimited

Caution: On most platforms, Oracle automatically sets the value of CPU_COUNT to the number of CPUs available to your Oracle instance. Do not change the value of CPU_COUNT.

CPU_COUNT specifies the number of CPUs available to Oracle. On single-CPU computers, the value of CPU_COUNT is 1.

CREATE_BITMAP_AREA_SIZE

Parameter type	Integer
Default value	8388608 (8 MB)
Parameter class	Static
Range of values	Operating system-dependent

Note: Oracle does not recommend using the CREATE_BITMAP_AREA_SIZE parameter unless the instance is configured with the shared server option. Oracle recommends that you enable automatic sizing of SQL working areas by setting PGA_AGGREGATE_TARGET instead. CREATE_BITMAP_AREA_SIZE is retained for backward compatibility.

CREATE_BITMAP_AREA_SIZE is relevant only for systems containing bitmap indexes. It specifies the amount of memory (in bytes) allocated for bitmap creation. The default value is 8 MB. A larger value may speed up index creation.

CREATE_STORED_OUTLINES

Syntax:

```
CREATE_STORED_OUTLINES = {TRUE | FALSE | category_name} [NOOVERRIDE]
```

The CREATE_STORED_OUTLINES parameter determines whether Oracle should automatically create and store an outline for each query submitted on the system. CREATE_STORED_OUTLINES is not an initialization parameter.

- TRUE enables automatic outline creation for subsequent queries in the system. These outlines receive a unique system-generated name and are stored in the DEFAULT category. If a particular query already has an outline defined for it in

the DEFAULT category, then that outline will remain and a new outline will not be created.

- FALSE disables automatic outline creation for the system. This is the default.
- *category_name* has the same behavior as TRUE except that any outline created in the system is stored in the *category_name* category.
- NOOVERRIDE specifies that this system setting will not override the setting for any session in which this parameter was explicitly set. If you do not specify NOOVERRIDE, then this setting takes effect in all sessions.

CURSOR_SHARING

Parameter type	String
Syntax	CURSOR_SHARING = {SIMILAR | EXACT | FORCE}
Default value	EXACT
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

CURSOR_SHARING determines what kind of SQL statements can share the same cursors.

CURSOR_SPACE_FOR_TIME

Parameter type	Boolean
Default value	false
Parameter class	Static
Range of values	true | false

CURSOR_SPACE_FOR_TIME lets you use more space for cursors in order to save time. It affects both the shared SQL area and the client's private SQL area.

DB_*n*K_CACHE_SIZE

Parameter type	Big integer
Syntax	DB_[2 | 4 | 8 | 16 | 32]K_CACHE_SIZE = *integer* [K | M | G]
Default value	0 (additional block size caches are not configured by default)

Parameter class	Dynamic: ALTER SYSTEM
Range of values	Minimum: the granule size Maximum: operating system-dependent

DB_*n*K_CACHE_SIZE (where *n* = 2, 4, 8, 16, 32) specifies the size of the cache for the *n*K buffers. You can set this parameter only when DB_BLOCK_SIZE has a value other than *n*K. For example, if DB_BLOCK_SIZE=4096, then it is illegal to specify the parameter DB_4K_CACHE_SIZE (because the size for the 4 KB block cache is already specified by DB_CACHE_SIZE).

DB_BLOCK_BUFFERS

Parameter type	Integer
Default value	Derived: 48 MB / DB_BLOCK_SIZE
Parameter class	Static
Range of values	50 to an operating system-specific maximum
Real Application Clusters	Multiple instances can have different values, and you can change the values as needed.

> **Note:** This parameter is deprecated in favor of the DB_CACHE_SIZE parameter. Oracle recommends that you use DB_CACHE_SIZE instead. Also, DB_BLOCK_BUFFERS cannot be combined with the new dynamic DB_CACHE_SIZE parameter; combining these parameters in the same parameter file will produce an error. DB_BLOCK_BUFFERS is retained for backward compatibility.

DB_BLOCK_BUFFERS specifies the number of database buffers in the buffer cache. It is one of several parameters that contribute to the total memory requirements of the SGA of an instance.

DB_BLOCK_CHECKING

Parameter type	Boolean
Default value	false
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

Range of values `true | false`

`DB_BLOCK_CHECKING` controls whether Oracle performs block checking for data blocks. When this parameter is set to `true`, Oracle performs block checking for all data blocks. When it is set to `false`, Oracle does not perform block checking for blocks in the user tablespaces. However, block checking for the `SYSTEM` tablespace is always turned on.

DB_BLOCK_CHECKSUM

Parameter type Boolean

Default value `true`

Parameter class Dynamic: `ALTER SYSTEM`

Range of values `true | false`

`DB_BLOCK_CHECKSUM` determines whether DBW*n* and the direct loader will calculate a **checksum** (a number calculated from all the bytes stored in the block) and store it in the cache header of every data block when writing it to disk. Checksums are verified when a block is read-only if this parameter is `true` and the last write of the block stored a checksum. In addition, Oracle gives every log block a checksum before writing it to the current log.

DB_BLOCK_SIZE

Parameter type Integer

Default value `2048`

Parameter class Static

Range of values 2048 to 32768, but your operating system may have a narrower range

Real Application Clusters You must set this parameter for every instance, and multiple instances must have the same value.

Caution: Set this parameter at the time of database creation. Do not alter it afterward.

DB_BLOCK_SIZE specifies the size (in bytes) of Oracle database blocks. Typical values are 2048 and 4096. The value for DB_BLOCK_SIZE in effect at the time you create the database determines the size of the blocks. The value must remain set to its initial value.

DB_CACHE_ADVICE

Parameter type	String
Syntax	DB_CACHE_ADVICE = {ON | READY | OFF}
Default value	If STATISTICS_LEVEL is set to TYPICAL or ALL, then ON If STATISTICS_LEVEL is set to BASIC, then OFF
Parameter class	Dynamic: ALTER SYSTEM

DB_CACHE_ADVICE enables or disables statistics gathering used for predicting behavior with different cache sizes through the V$DB_CACHE_ADVICE performance view.

DB_CACHE_SIZE

Parameter type	Big integer
Syntax	DB_CACHE_SIZE = *integer* [K | M | G]
Default value	48 MB, rounded up to the nearest granule size
Parameter class	Dynamic: ALTER SYSTEM

DB_CACHE_SIZE specifies the size of the DEFAULT buffer pool for buffers with the primary block size (the block size defined by the DB_BLOCK_SIZE parameter).

DB_CREATE_FILE_DEST

Parameter type	String
Syntax	DB_CREATE_FILE_DEST = *directory*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

DB_CREATE_FILE_DEST sets the default location for Oracle-managed datafiles. This location is also used as the default for Oracle-managed control files and online redo logs if DB_CREATE_ONLINE_LOG_DEST_*n* is not specified.

DB_CREATE_ONLINE_LOG_DEST_*n*

Parameter type	String
Syntax	DB_CREATE_ONLINE_LOG_DEST_[1 | 2 | 3 | 4 | 5] = *directory*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

DB_CREATE_ONLINE_LOG_DEST_*n* (where *n* = 1, 2, 3, ... 5) sets the default location for Oracle-managed control files and online redo logs.

DB_DOMAIN

Parameter type	String
Syntax	DB_DOMAIN = *domain_name*
Default value	There is no default value.
Parameter class	Static
Range of values	Any legal string of name components, separated by periods and up to 128 characters long (including the periods). This value cannot be NULL.
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have the same value.

In a distributed database system, DB_DOMAIN specifies the logical location of the database within the network structure. You should set this parameter if this database is or ever will be part of a distributed system. The value consists of the extension components of a global database name, consisting of valid identifiers, separated by periods. Oracle Corporation recommends that you specify DB_DOMAIN as a unique string for all databases in a domain.

DB_FILE_MULTIBLOCK_READ_COUNT

Parameter type	Integer

Default value 8

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM

Range of values Operating system-dependent

DB_FILE_MULTIBLOCK_READ_COUNT is one of the parameters you can use to minimize I/O during table scans. It specifies the maximum number of blocks read in one I/O operation during a sequential scan. The total number of I/Os needed to perform a full table scan depends on such factors as the size of the table, the multiblock read count, and whether parallel execution is being utilized for the operation.

DB_FILE_NAME_CONVERT

Parameter type String

Syntax

```
DB_FILE_NAME_CONVERT = [()'string1' ,
'string2' , 'string3' , 'string4' , ...[])]
```

Where:

- string1 is the pattern of the primary database filename
- string2 is the pattern of the standby database filename
- string3 is the pattern of the primary database filename
- string4 is the pattern of the standby database filename

You can use as many pairs of primary and standby replacement strings as required. You can use single or double quotation marks. The parentheses are optional.

Following are example settings that are acceptable:

```
DB_FILE_NAME_CONVERT =
('/dbs/t1/','/dbs/t1/s_','dbs/t2/
','dbs/t2/s_')
```

Default value None

Parameter class Static

DB_FILE_NAME_CONVERT is useful for creating a duplicate database for recovery purposes. It converts the filename of a new datafile on the primary database to a filename on the standby database. If you add a datafile to the primary database, you must add a corresponding file to the standby database. When the standby database

is updated, this parameter converts the datafile name on the primary database to the datafile name on the standby database. The file on the standby database must exist and be writable, or the recovery process will halt with an error.

DB_FILES

Parameter type	Integer
Default value	200
Parameter class	Static
Range of values	Minimum: the largest among the absolute file numbers of the datafiles in the database
	Maximum: operating system-dependent
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have the same value.

DB_FILES specifies the maximum number of database files that can be opened for this database. The maximum valid value is the maximum number of files, subject to operating system constraint, that will ever be specified for the database, including files to be added by ADD DATAFILE statements.

DB_KEEP_CACHE_SIZE

Parameter type	Big integer
Syntax	DB_KEEP_CACHE_SIZE = *integer* [K | M | G]
Default value	0 (KEEP cache is not configured by default)
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Minimum: the granule size
	Maximum: operating system-dependent

DB_KEEP_CACHE_SIZE specifies the size of the KEEP buffer pool. The size of the buffers in the KEEP buffer pool is the primary block size (the block size defined by the DB_BLOCK_SIZE parameter).

DB_NAME

Parameter type	String

Syntax	DB_NAME = *database_name*
Default value	There is no default value.
Parameter class	Static
Real Application Clusters	You must set this parameter for every instance. Multiple instances must have the same value, or the same value must be specified in the STARTUP OPEN SQL*Plus statement or the ALTER DATABASE MOUNT SQL statement.

DB_NAME specifies a database identifier of up to 8 characters. If specified, it must correspond to the name specified in the CREATE DATABASE statement. Although the use of DB_NAME is optional, you should generally set it before issuing the CREATE DATABASE statement, and then reference it in that statement.

DB_RECYCLE_CACHE_SIZE

Parameter type	Big integer
Syntax	DB_RECYCLE_CACHE_SIZE = *integer* [K | M | G]
Default value	0 (RECYCLE cache is not configured by default)
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Minimum: the granule size Maximum: operating system-dependent

DB_RECYCLE_CACHE_SIZE specifies the size of the RECYCLE buffer pool. The size of the buffers in the RECYCLE pool is the primary block size (the block size defined by the DB_BLOCK_SIZE parameter).

DB_WRITER_PROCESSES

Parameter type	Integer
Default value	1
Parameter class	Static
Range of values	1 to 20

DB_WRITER_PROCESSES is useful for systems that modify data heavily. It specifies the initial number of database writer processes for an instance.

DBLINK_ENCRYPT_LOGIN

Parameter type	Boolean
Default value	`false`
Parameter class	Static
Range of values	`true	false`

When you attempt to connect to a database using a password, Oracle encrypts the password before sending it to the database. `DBLINK_ENCRYPT_LOGIN` specifies whether or not attempts to connect to other Oracle servers through database links should use encrypted passwords.

DBWR_IO_SLAVES

Parameter type	Integer
Default value	`0`
Parameter class	Static
Range of values	0 to operating system-dependent

`DBWR_IO_SLAVES` is relevant only on systems with only one database writer process (`DBW0`). It specifies the number of I/O server processes used by the `DBW0` process. The `DBW0` process and its server processes always write to disk. By default, the value is `0` and I/O server processes are not used.

DG_BROKER_CONFIG_FILE*n*

Parameter type	String
Syntax	`DG_BROKER_CONFIG_FILE[1	2] = filename`
Default value	Operating system-dependent
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	One filename

`DG_BROKER_CONFIG_FILE`*n* (where *n* = 1, 2) specifies the names for the Data Guard broker configuration files.

DG_BROKER_START

Parameter type Boolean

Default value `false`

Parameter class Dynamic: `ALTER SYSTEM`

Range of values `true | false`

`DG_BROKER_START` enables Oracle to determine whether or not the DMON (Data Guard broker) process should be started. DMON is a non-fatal Oracle background process and exists as long as the instance exists, whenever this parameter is set to `true`.

DISK_ASYNCH_IO

Parameter type Boolean

Default value `true`

Parameter class Static

Range of values `true | false`

`DISK_ASYNCH_IO` controls whether I/O to datafiles, control files, and logfiles is asynchronous (that is, whether parallel server processes can overlap I/O requests with CPU processing during table scans). If your platform supports asynchronous I/O to disk, Oracle Corporation recommends that you leave this parameter set to its default value. However, if the asynchronous I/O implementation is not stable, you can set this parameter to `false` to disable asynchronous I/O. If your platform does not support asynchronous I/O to disk, this parameter has no effect.

DISPATCHERS

Parameter type String

Syntax `DISPATCHERS = '`*`dispatch_clause`*`'`

dispatch_clause::=

```
(PROTOCOL = protocol) |
(ADDRESS = address) |
(DESCRIPTION = description )
[options_clause]
```

options_clause::=

```
(DISPATCHERS = integer |
 SESSIONS = integer |
 CONNECTIONS = integer |
 TICKS = seconds |
 POOL = {1 | ON | YES | TRUE | BOTH |
   ({IN | OUT} = ticks) | 0 | OFF | NO |
FALSE |
   ticks} |
 MULTIPLEX = {1 | ON | YES | TRUE |
   0 | OFF | NO | FALSE | BOTH | IN | OUT} |
 LISTENER = tnsname |
 SERVICE = service |
 INDEX = integer)
```

Default value There is no default value.

Parameter class Dynamic: ALTER SYSTEM

DISPATCHERS configures dispatcher processes in the shared server architecture. The parsing software supports a name-value syntax to enable the specification of attributes in a position-independent case-insensitive manner. For example:

```
DISPATCHERS = "(PROTOCOL=TCP)(DISPATCHERS=3)"
```

DISTRIBUTED_LOCK_TIMEOUT

Parameter type Integer

Default value	60
Parameter class	Static
Range of values	1 to unlimited

DISTRIBUTED_LOCK_TIMEOUT specifies the amount of time (in seconds) for distributed transactions to wait for locked resources.

DML_LOCKS

Parameter type	Integer
Default value	Derived: 4 * TRANSACTIONS
Parameter class	Static
Range of values	20 to unlimited; a setting of 0 disables enqueues
Real Application Clusters	You must set this parameter for every instance, and all instances must have positive values or all must be 0.

A **DML lock** is a lock obtained on a table that is undergoing a DML operation (insert, update, delete). DML_LOCKS specifies the maximum number of DML locks—one for each table modified in a transaction. The value should equal the grand total of locks on tables currently referenced by all users. For example, if three users are modifying data in one table, then three entries would be required. If three users are modifying data in two tables, then six entries would be required.

> **Note:** You can set this parameter using ALTER SYSTEM only if you have started up the database using a server parameter file (spfile), and you must specify SCOPE = SPFILE.

DRS_START

Parameter type	Boolean
Default value	false
Parameter class	Dynamic: ALTER SYSTEM
Range of values	true | false

> **Note:** This parameter is deprecated in favor of the DG_BROKER_START parameter. Oracle recommends that you use DG_BROKER_START instead. DRS_START is retained for backward compatibility only.

DRS_START enables Oracle to determine whether or not the DMON (Data Guard broker) process should be started. DMON is a non-fatal Oracle background process and exists as long as the instance exists, whenever this parameter is set to `true`.

ENQUEUE_RESOURCES

Parameter type Integer

Default value Derived from SESSIONS parameter

Parameter class Static

Range of values 10 to unlimited

ENQUEUE_RESOURCES sets the number of resources that can be concurrently locked by the lock manager. An **enqueue** is a sophisticated locking mechanism that permits several concurrent processes to share known resources to varying degrees. Any object that can be used concurrently can be protected with enqueues. For example, Oracle allows varying levels of sharing on tables: two processes can lock a table in share mode or in share update mode.

> **Note:** You can set this parameter using ALTER SYSTEM only if you have started up the database using a server parameter file (spfile), and you must specify SCOPE = SPFILE.

EVENT

Parameter type String

Default value There is no default value.

Parameter class Static

EVENT is a parameter used only to debug the system. Do not alter the value of this parameter except under the supervision of Oracle Support Services staff.

FAL_CLIENT

Parameter type	String
Syntax	FAL_CLIENT = *string*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM

FAL_CLIENT specifies the FAL (fetch archive log) client name that is used by the FAL service, configured through the FAL_SERVER parameter, to refer to the FAL client. The value is an Oracle Net service name, which is assumed to be configured properly on the FAL server system to point to the FAL client (standby database).

FAL_SERVER

Parameter type	String
Syntax	FAL_SERVER = *string*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM

FAL_SERVER specifies the FAL (fetch archive log) server for a standby database. The value is an Oracle Net service name, which is assumed to be configured properly on the standby database system to point to the desired FAL server.

FAST_START_IO_TARGET

Parameter type	Integer
Default value	All the buffers in the cache
Parameter class	Dynamic: ALTER SYSTEM
Range of values	1000 to all buffers in the cache. A setting of 0 disables limiting recovery I/Os.
Real Application Clusters	Multiple instances can have different values, and you can change the values at runtime.

Note: This parameter is deprecated in favor of the FAST_START_MTTR_TARGET parameter. Oracle recommends that you use FAST_START_MTTR_TARGET instead. FAST_START_IO_TARGETis retained for backward compatibility only.

FAST_START_IO_TARGET (available only with the Oracle Enterprise Edition) specifies the number of I/Os that should be needed during crash or instance recovery.

FAST_START_MTTR_TARGET

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to 3600 seconds
Real Application Clusters	Multiple instances can have different values, and you can change the values at runtime.

FAST_START_MTTR_TARGET enables you to specify the number of seconds the database takes to perform crash recovery of a single instance. When specified, FAST_START_MTTR_TARGET

- Is overridden by FAST_START_IO_TARGET
- Is overridden by LOG_CHECKPOINT_INTERVAL

FAST_START_PARALLEL_ROLLBACK

Parameter type	String
Syntax	FAST_START_PARALLEL_ROLLBACK = {HI | LO | FALSE}
Default value	LOW
Parameter class	Dynamic: ALTER SYSTEM

FAST_START_PARALLEL_ROLLBACK determines the maximum number of processes that can exist for performing parallel rollback. This parameter is useful on systems in which some or all of the transactions are long running.

FILE_MAPPING

Parameter type Boolean

Default value `false`

Parameter class Dynamic: `ALTER SYSTEM`

Range of values `true | false`

`FILE_MAPPING` enables or disables file mapping. The FMON background process will be started to manage the mapping information when file mapping is enabled.

FILESYSTEMIO_OPTIONS

Parameter type String

Syntax `FILESYSTEMIO_OPTIONS = {none | setall | directIO | asynch}`

Default value There is no default value.

Parameter class Dynamic: `ALTER SESSION`, `ALTER SYSTEM`

`FILESYSTEMIO_OPTIONS` specifies I/O operations for file system files.

FIXED_DATE

Parameter type String

Syntax `FIXED_DATE = YYYY-MM-DD-HH24:MI:SS` (or the default Oracle date format)

Default value There is no default value.

Parameter class Dynamic: `ALTER SYSTEM`

`FIXED_DATE` enables you to set a constant date that `SYSDATE` will always return instead of the current date. This parameter is useful primarily for testing. The value can be in the format shown above or in the default Oracle date format, without a time.

GC_FILES_TO_LOCKS

Parameter type String

Syntax	GC_FILES_TO_LOCKS = '{*file_list*=*lock_count*[!*blocks*][EACH][:...]}' Spaces are not allowed within the quotation marks.
Default value	There is no default value.
Parameter class	Static
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have identical values. To change the value, you must shut down all instances in the cluster, change the value for each instance, and then start up each instance.

Note: Setting this parameter to any value other than the default will disable Cache Fusion processing in Oracle9*i* Real Application Clusters.

GC_FILES_TO_LOCKS is an Oracle9*i* Real Application Clusters parameter that has no effect on an instance running in exclusive mode. It controls the mapping of pre-release 9.0.1 parallel cache management (PCM) locks to datafiles.

GLOBAL_CONTEXT_POOL_SIZE

Parameter type	String
Default value	1 MB
Parameter class	Static
Range of values	Any integer value in MB

GLOBAL_CONTEXT_POOL_SIZE specifies the amount of memory to allocate in the SGA for storing and managing global application context.

GLOBAL_NAMES

Parameter type	Boolean
Default value	false
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

Range of values `true | false`

GLOBAL_NAMES specifies whether a database link is required to have the same name as the database to which it connects.

HASH_AREA_SIZE

Parameter type Integer

Default value Derived: 2 * SORT_AREA_SIZE

Parameter class Dynamic: ALTER SESSION

Range of values 0 to operating system-dependent

> **Note:** Oracle does not recommend using the HASH_AREA_SIZE parameter unless the instance is configured with the shared server option. Oracle recommends that you enable automatic sizing of SQL working areas by setting PGA_AGGREGATE_TARGET instead. HASH_AREA_SIZE is retained for backward compatibility.

HASH_AREA_SIZE is relevant to parallel execution operations and to the query portion of DML or DDL statements. It specifies the maximum amount of memory, in bytes, to be used for hash joins.

HASH_JOIN_ENABLED

Parameter type Boolean

Default value `true`

Parameter class Dynamic: ALTER SESSION

Range of values `true | false`

HASH_JOIN_ENABLED specifies whether the optimizer should consider using a hash join as a join method. If set to `false`, then hashing is not available as a join method. If set to `true`, then the optimizer compares the cost of a hash join with other types of joins, and chooses hashing if it gives the best cost. Oracle Corporation recommends that you set this parameter to `true` for all data warehousing applications.

HI_SHARED_MEMORY_ADDRESS

Parameter type	Integer
Default value	`0`
Parameter class	Static

`HI_SHARED_MEMORY_ADDRESS` specifies the starting address at runtime of the system global area (SGA). It is ignored on platforms that specify the SGA's starting address at linktime.

HS_AUTOREGISTER

Parameter type	Boolean
Default value	`true`
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	`true | false`

`HS_AUTOREGISTER` enables or disables automatic self-registration of Heterogeneous Services (HS) agents. When enabled, information is uploaded into the server's data dictionary to describe a previously unknown agent class or a new agent version.

IFILE

Parameter type	Parameter file
Syntax	`IFILE = parameter_file_name`
Default value	There is no default value.
Parameter class	Static
Range of values	Valid parameter filenames
Real Application Clusters	Multiple instances can have different values.

Use `IFILE` to embed another parameter file within the current parameter file. For example:

```
IFILE = COMMON.ORA
```

INSTANCE_GROUPS

Parameter type	String
Syntax	`INSTANCE_GROUPS = group_name [, group_name ... ]`
Default value	There is no default value.
Parameter class	Static
Range of values	One or more instance group names, separated by commas
Real Application Clusters	Multiple instances can have different values.

`INSTANCE_GROUPS` is an Oracle9*i* Real Application Clusters parameter that you can specify only in parallel mode. Used in conjunction with the `PARALLEL_INSTANCE_GROUP` parameter, it lets you restrict parallel query operations to a limited number of instances.

INSTANCE_NAME

Parameter type	String
Syntax	`INSTANCE_NAME = instance_id`
Default value	The instance's SID **Note:** The SID identifies the instance's shared memory on a host, but may not uniquely distinguish this instance from other instances.
Parameter class	Static
Range of values	Any alphanumeric characters

In an Oracle9*i* Real Application Clusters environment, multiple instances can be associated with a single database service. Clients can override Oracle's connection load balancing by specifying a particular instance by which to connect to the database. `INSTANCE_NAME` specifies the unique name of this instance.

INSTANCE_NUMBER

Parameter type	Integer

Default value	Lowest available number; derived from instance start up order and INSTANCE_NUMBER value of other instances. If not configured for Oracle9*i* Real Application Clusters, then 0.
Parameter class	Static
Range of values	1 to maximum number of instances specified when the database was created
Real Application Clusters	You must set this parameter for every instance, and all instances must have different values.

INSTANCE_NUMBER is an Oracle9*i* Real Application Clusters parameter that can be specified in parallel mode or exclusive mode. It specifies a unique number that maps the instance to one free list group for each database object created with storage parameter FREELIST GROUPS.

JAVA_MAX_SESSIONSPACE_SIZE

Parameter type	Integer
Default value	0
Parameter class	Static
Range of values	0 to 4 GB

Java session space is the memory that holds Java state from one database call to another. JAVA_MAX_SESSIONSPACE_SIZE specifies (in bytes) the maximum amount of session space made available to a Java program executing in the server. When a user's session-duration Java state attempts to exceed this amount, the Java virtual machine kills the session with an out-of-memory failure.

JAVA_POOL_SIZE

Parameter type	Big integer
Syntax	LARGE_POOL_SIZE = *integer* [K | M | G]
Default value	24 MB, rounded up to the nearest granule size
Parameter class	Static

Range of values	Minimum: the granule size Maximum: operating system-dependent

JAVA_POOL_SIZE specifies the size (in bytes) of the Java pool, from which the Java memory manager allocates most Java state during runtime execution. This memory includes the shared in-memory representation of Java method and class definitions, as well as the Java objects that are migrated to the Java session space at end-of-call.

JAVA_SOFT_SESSIONSPACE_LIMIT

Parameter type	Integer
Default value	0
Parameter class	Static
Range of values	0 to 4 GB

Java session space is the memory that holds Java state from one database call to another. JAVA_SOFT_SESSIONSPACE_LIMIT specifies (in bytes) a **soft limit** on Java memory usage in a session, as a means to warn you if a user's session-duration Java state is using too much memory. When a user's session-duration Java state exceeds this size, Oracle generates a warning that goes into the trace files.

JOB_QUEUE_PROCESSES

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to 1000
Real Application Clusters	Multiple instances can have different values.

JOB_QUEUE_PROCESSES specifies the maximum number of processes that can be created for the execution of jobs. It specifies the number of job queue processes per instance (J000, ... J999). Replication uses job queues for data refreshes. Advanced queuing uses job queues for message propagation. You can create user job requests through the DBMS_JOB package.

LARGE_POOL_SIZE

Parameter type	Big integer
Syntax	LARGE_POOL_SIZE = *integer* [K | M | G]
Default value	0 if both of the following are true: ■ The pool is not required by parallel execution ■ DBWR_IO_SLAVES is not set Otherwise, derived from the values of PARALLEL_MAX_SERVERS, PARALLEL_THREADS_PER_CPU, CLUSTER_DATABASE_INSTANCES, DISPATCHERS, and DBWR_IO_SLAVES.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	300 KB to at least 2 GB (actual maximum is operating system-specific)

LARGE_POOL_SIZE lets you specify the size (in bytes) of the large pool allocation heap. The large pool allocation heap is used in shared server systems for session memory, by parallel execution for message buffers, and by backup processes for disk I/O buffers. (Parallel execution allocates buffers out of the large pool only when PARALLEL_AUTOMATIC_TUNING is set to true.)

LICENSE_MAX_SESSIONS

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to number of session licenses
Real Application Clusters	Multiple instances can have different values, but the total for all instances mounting a database should be less than or equal to the total number of sessions licensed for that database.

LICENSE_MAX_SESSIONS specifies the maximum number of concurrent user sessions allowed. When this limit is reached, only users with the RESTRICTED SESSION privilege can connect to the database. Users who are not able to connect

receive a warning message indicating that the system has reached maximum capacity.

LICENSE_MAX_USERS

Parameter type	Integer
Default value	`0`
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	0 to number of user licenses
Real Application Clusters	Multiple instances should have the same values. If different instances specify different values for this parameter, then the value of the first instance to mount the database takes precedence.

`LICENSE_MAX_USERS` specifies the maximum number of users you can create in the database. When you reach this limit, you cannot create more users. You can, however, increase the limit.

Restriction on LICENSE_MAX_USERS: You cannot reduce the limit on users below the current number of users created for the database.

> **See Also:** "Changing Licensing Parameters: Examples" on page 10-123

LICENSE_SESSIONS_WARNING

Parameter type	Integer
Default value	`0`
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	0 to value of `LICENSE_MAX_SESSIONS` parameter
Real Application Clusters	Multiple instances can have different values.

`LICENSE_SESSIONS_WARNING` specifies a warning limit on the number of concurrent user sessions. When this limit is reached, additional users can connect, but Oracle writes a message in the alert file for each new connection. Users with `RESTRICTED SESSION` privilege who connect after the limit is reached receive a warning message stating that the system is nearing its maximum capacity.

LOCAL_LISTENER

Parameter type	String
Syntax	LOCAL_LISTENER = *network_name*
Default value	(ADDRESS = (PROTOCOL=TCP)(HOST=)(PORT=1521))
Parameter class	Dynamic: ALTER SYSTEM

LOCAL_LISTENER specifies a network name that resolves to an address or address list of Oracle Net local listeners (that is, listeners that are running on the same machine as this instance). The address or address list is specified in the TNSNAMES.ORA file or other address repository as configured for your system.

LOCK_NAME_SPACE

Parameter type	String
Syntax	LOCK_NAME_SPACE = *namespace*
Default value	There is no default value.
Parameter class	Static
Range of values	Up to 8 alphanumeric characters. No special characters allowed.

LOCK_NAME_SPACE specifies the namespace that the distributed lock manager (DLM) uses to generate lock names. Consider setting this parameter if a standby or clone database has the same database name on the same cluster as the primary database.

LOCK_SGA

Parameter type	Boolean
Default value	false
Parameter class	Static
Range of values	true | false

LOCK_SGA locks the entire SGA into physical memory. It is usually advisable to lock the SGA into real (physical) memory, especially if the use of virtual memory would include storing some of the SGA using disk space. This parameter is ignored on platforms that do not support it.

LOG_ARCHIVE_DEST

Parameter type	String
Syntax	LOG_ARCHIVE_DEST = *filespec*
Default value	Null
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid path or device name, except raw partitions
Real Application Clusters	Multiple instances can have different values.

> **Note:** For Enterprise Edition users, this parameter has been deprecated in favor of the LOG_ARCHIVE_DEST_*n* parameters. If Oracle Enterprise Edition is not installed or it is installed, but you have not specified any LOG_ARCHIVE_DEST_*n* parameters, this parameter is valid.

LOG_ARCHIVE_DEST is applicable only if you are running the database in ARCHIVELOG mode or are recovering a database from archived redo logs. LOG_ARCHIVE_DEST is incompatible with the LOG_ARCHIVE_DEST_*n* parameters, and must be defined as the null string ("") or (' ') when any LOG_ARCHIVE_DEST_*n* parameter has a value other than a null string. Use a text string to specify the default location and root of the disk file or tape device when archiving redo log files. (Archiving to tape is not supported on all operating systems.) The value cannot be a raw partition.

LOG_ARCHIVE_DEST_*n*

Parameter type	String

Syntax

```
LOG_ARCHIVE_DEST_[1 | 2 | 3 | 4 | 5 | 6 | 7 |
8 | 9 | 10] =

   { null_string} |
     { LOCATION=path_name | SERVICE=service_name }
     [ { MANDATORY | OPTIONAL } ]
     [ REOPEN[=seconds] | NOREOPEN ]
     [ DELAY[=minutes] | NODELAY ]
     [ REGISTER[=template] | NOREGISTER ]
     [ TEMPLATE=template] | NOTEMPLATE ]
     [ ALTERNATE=destination | NOALTERNATE ]
     [ DEPENDENCY=destination | NODEPENDENCY ]
     [ MAX_FAILURE=count | NOMAX_FAILURE ]
     [ QUOTA_SIZE=blocks | NOQUOTA_SIZE ]
     [ QUOTA_USED=blocks | NOQUOTA_USED ]
     [ ARCH | LGWR ]
     [ SYNC[=PARALLEL|NOPARALLEL] | ASYNC[=blocks] ]
     [ AFFIRM | NOAFFIRM ]
     [ NET_TIMEOUT=seconds | NONET_TIMEOUT ]
   }
```

Default value There is no default value.

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM

Note: This parameter is valid only if you have installed Oracle Enterprise Edition. You may continue to use LOG_ARCHIVE_DEST if you have installed Oracle Enterprise Edition. However, you cannot use both LOG_ARCHIVE_DEST_*n* and LOG_ARCHIVE_DEST, as they are not compatible.

The LOG_ARCHIVE_DEST_*n* parameters (where *n* = 1, 2, 3, ... 10) define up to ten archive log destinations. The parameter integer suffix is defined as the **handle** displayed by the V$ARCHIVE_DEST dynamic performance view.

LOG_ARCHIVE_DEST_STATE_*n*

Parameter type String

Syntax

```
LOG_ARCHIVE_DEST_STATE_n = {alternate | reset
| defer | enable}
```

Default value enable

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM

The LOG_ARCHIVE_DEST_STATE_*n* parameters (where *n* = 1, 2, 3, ... 10) specify the availability state of the corresponding destination. The parameter suffix (1 through 10) specifies one of the ten corresponding LOG_ARCHIVE_DEST_*n* destination parameters.

LOG_ARCHIVE_DUPLEX_DEST

Parameter type String

Syntax LOG_ARCHIVE_DUPLEX_DEST = *filespec*

Default value There is no default value.

Parameter class Dynamic: ALTER SYSTEM

Range of values Either a null string or any valid path or device name, except raw partitions

---

Note: If you are using Oracle Enterprise Edition, this parameter is deprecated in favor of the LOG_ARCHIVE_DEST_*n* parameters. If Oracle Enterprise Edition is not installed or it is installed but you have not specified any LOG_ARCHIVE_DEST_*n* parameters, this parameter is valid.

LOG_ARCHIVE_DUPLEX_DEST is similar to the initialization parameter LOG_ARCHIVE_DEST. This parameter specifies a second archive destination: the **duplex** archive destination. This duplex archive destination can be either a must-succeed or a best-effort archive destination, depending on how many archive destinations must succeed (as specified in the LOG_ARCHIVE_MIN_SUCCEED_DEST parameter).

LOG_ARCHIVE_FORMAT

Parameter type String

Syntax LOG_ARCHIVE_FORMAT = *filename*

Default value Operating system-dependent

Parameter class Static

Range of values Any string that resolves to a valid filename

Real Application Clusters	Multiple instances can have different values, but identical values are recommended.

LOG_ARCHIVE_FORMAT is applicable only if you are using the redo log in ARCHIVELOG mode. Use a text string and variables to specify the default filename format when archiving redo log files. The string generated from this format is appended to the string specified in the LOG_ARCHIVE_DEST parameter.

LOG_ARCHIVE_MAX_PROCESSES

Parameter type	Integer
Default value	1
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any integer from 1 to 10

LOG_ARCHIVE_MAX_PROCESSES specifies the number of archiver background processes (ARC0 through ARC9) Oracle initially invokes.

LOG_ARCHIVE_MIN_SUCCEED_DEST

Parameter type	Integer
Default value	1
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM
Range of values	1 to 10 if you are using LOG_ARCHIVE_DEST_*n* 1 or 2 if you are using LOG_ARCHIVE_DEST and LOG_ARCHIVE_DUPLEX_DEST

LOG_ARCHIVE_MIN_SUCCEED_DEST defines the minimum number of destinations that must succeed in order for the online logfile to be available for reuse.

LOG_ARCHIVE_START

Parameter type	Boolean
Default value	false
Parameter class	Static
Range of values	true | false

Real Application Clusters	Multiple instances can have different values.

LOG_ARCHIVE_START is applicable only when you use the redo log in ARCHIVELOG mode. It indicates whether archiving should be automatic or manual when the instance starts up.

LOG_ARCHIVE_TRACE

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0, 1, 2, 4, 8, 16, 32, 64, 128
Real Application Clusters	Multiple instances can have different values.

LOG_ARCHIVE_TRACE controls output generated by the archivelog process.

LOG_BUFFER

Parameter type	Integer
Default value	512 KB or 128 KB * CPU_COUNT, whichever is greater
Parameter class	Static
Range of values	Operating system-dependent

LOG_BUFFER specifies the amount of memory (in bytes) that Oracle uses when buffering redo entries to a redo log file. Redo log entries contain a record of the changes that have been made to the database block buffers. The LGWR process writes redo log entries from the log buffer to a redo log file.

LOG_CHECKPOINT_INTERVAL

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Unlimited

Real Application Clusters	Multiple instances can have different values.

LOG_CHECKPOINT_INTERVAL specifies the frequency of checkpoints in terms of the number of redo log file blocks that can exist between an incremental checkpoint and the last block written to the redo log. This number refers to physical operating system blocks, not database blocks.

LOG_CHECKPOINT_TIMEOUT

Parameter type	Integer
Default value	1800
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to unlimited
Real Application Clusters	Multiple instances can have different values.

LOG_CHECKPOINT_TIMEOUT specifies (in seconds) the amount of time that has passed since the incremental checkpoint at the position where the last write to the redo log (sometimes called the **tail of the log**) occurred. This parameter also signifies that no buffer will remain dirty (in the cache) for more than *integer* seconds.

LOG_CHECKPOINTS_TO_ALERT

Parameter type	Boolean
Default value	false
Parameter class	Dynamic: ALTER SYSTEM
Range of values	true | false

LOG_CHECKPOINTS_TO_ALERT lets you log your checkpoints to the alert file. Doing so is useful for determining whether checkpoints are occurring at the desired frequency.

LOG_FILE_NAME_CONVERT

Parameter type	String

Syntax `LOG_FILE_NAME_CONVERT = [()'string1' , 'string2' , 'string3' , 'string4' , ...[])]`

Where:

- `string1` is the pattern of the primary database filename
- `string2` is the pattern of the standby database filename
- `string3` is the pattern of the primary database filename
- `string4` is the pattern of the standby database filename

You can use as many pairs of primary and standby replacement strings as required. You can use single or double quotation marks. The parentheses are optional.

Following are example settings that are acceptable:

```
LOG_FILE_NAME_CONVERT=('/dbs/t1/','/dbs/t1/s_
','dbs/t2/ ','dbs/t2/s_')
```

Default value None

Parameter class Static

Range of values Character strings

`LOG_FILE_NAME_CONVERT` converts the filename of a new log file on the primary database to the filename of a log file on the standby database. If you add a log file to the primary database, you must add a corresponding file to the standby database.

LOG_PARALLELISM

Parameter type Integer

Default value `1`

Parameter class Static

Range of values `1` to `255`

`LOG_PARALLELISM` specifies the level of concurrency for redo allocation within Oracle.

LOGMNR_MAX_PERSISTENT_SESSIONS

Parameter type Integer

Default value	1
Parameter class	Static
Range of values	1 to LICENSE_MAX_SESSIONS

LOGMNR_MAX_PERSISTENT_SESSIONS enables you to specify the maximum number of persistent LogMiner mining sessions (which are LogMiner sessions that are backed up on disk) that are concurrently active when all sessions are mining redo logs generated by standalone instances. This pre-allocates 2*LOGMNR_MAX_PERSISTENT_SESSIONS MB of contiguous memory in the SGA for use by LogMiner.

MAX_COMMIT_PROPAGATION_DELAY

Parameter type	Integer
Default value	700
Parameter class	Static
Range of values	0 to 90000
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have identical values.

> **Caution:** Change this parameter only when it is absolutely necessary to see the most current version of the database when performing a query.

MAX_COMMIT_PROPAGATION_DELAY is an Oracle9*i* Real Application Clusters parameter. This initialization parameter should not be changed except under a limited set of circumstances specific to the cluster database.

MAX_DISPATCHERS

Parameter type	Integer
Default value	5
Parameter class	Static
Range of values	5 or the number of dispatchers configured, whichever is greater

MAX_DISPATCHERS specifies the maximum number of dispatcher processes allowed to be running simultaneously. The default value applies only if dispatchers have been configured for the system.

MAX_DUMP_FILE_SIZE

Parameter type	String
Syntax	MAX_DUMP_FILE_SIZE = {*integer* [K | M] | UNLIMITED}
Default value	UNLIMITED
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM
Range of values	0 to unlimited, or UNLIMITED

MAX_DUMP_FILE_SIZE specifies the maximum size of trace files (excluding the alert file). Change this limit if you are concerned that trace files may use too much space.

MAX_ENABLED_ROLES

Parameter type	Integer
Default value	20
Parameter class	Static
Range of values	0 to 148

MAX_ENABLED_ROLES specifies the maximum number of database roles that users can enable, including roles contained within other roles.

MAX_ROLLBACK_SEGMENTS

Parameter type	Integer
Default value	MAX(30, TRANSACTIONS/TRANSACTIONS_PER_ROLLBACK_SEGMENT)
Parameter class	Static
Range of values	2 to 65535

MAX_ROLLBACK_SEGMENTS specifies the maximum size of the rollback segment cache in the SGA. The number specified signifies the maximum number of rollback

segments that can be kept online (that is, status of ONLINE) simultaneously by one instance.

MTS Parameters

See "Shared Server Parameters" on page 10-110.

MAX_SHARED_SERVERS

Parameter type	Integer
Default value	Derived from SHARED_SERVERS (either 20 or 2*SHARED_SERVERS)
Parameter class	Static
Range of values	Operating system-dependent

MAX_SHARED_SERVERS specifies the maximum number of shared server processes allowed to be running simultaneously. If artificial deadlocks occur too frequently on your system, you should increase the value of MAX_SHARED_SERVERS.

NLS_CALENDAR

Parameter type	String
Syntax	NLS_CALENDAR = "*calendar_system*"
Default value	None
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid calendar format name

NLS_CALENDAR specifies which calendar system Oracle uses. It can have one of the following values:

- Arabic Hijrah
- English Hijrah
- Gregorian
- Japanese Imperial
- Persian
- ROC Official (Republic of China)

- Thai Buddha

NLS_COMP

Parameter type	String
Syntax	NLS_COMP = {BINARY | ANSI}
Default value	BINARY
Parameter class	Dynamic: ALTER SESSION

Normally, comparisons in the WHERE clause and in PL/SQL blocks is binary unless you specify the NLSSORT function. By setting NLS_COMP to ANSI, you indicate that comparisons in the WHERE clause and in PL/SQL blocks should use the linguistic sort specified in the NLS_SORT parameter. You must also define an index on the column for which you want linguistic sorts.

NLS_CURRENCY

Parameter type	String
Syntax	NLS_CURRENCY = *currency_symbol*
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid character string, with a maximum of 10 bytes (not including null)

NLS_CURRENCY specifies the string to use as the local currency symbol for the L number format element. The default value of this parameter is determined by NLS_TERRITORY.

NLS_DATE_FORMAT

Parameter type	String
Syntax	NLS_DATE_FORMAT = "*format*"
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid date format mask but not exceeding a fixed length

NLS_DATE_FORMAT specifies the default date format to use with the TO_CHAR and TO_DATE functions. The default value of this parameter is determined by NLS_TERRITORY.

NLS_DATE_LANGUAGE

Parameter type	String
Syntax	NLS_DATE_LANGUAGE = *language*
Default value	Derived from NLS_LANGUAGE
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid NLS_LANGUAGE value

NLS_DATE_LANGUAGE specifies the language to use for the spelling of day and month names and date abbreviations (a.m., p.m., AD, BC) returned by the TO_DATE and TO_CHAR functions.

NLS_DUAL_CURRENCY

Parameter type	String
Syntax	NLS_DUAL_CURRENCY = *currency_symbol*
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid format name up to 10 characters

NLS_DUAL_CURRENCY specifies the dual currency symbol (such as "Euro") for the territory. The default is the dual currency symbol defined in the territory of your current language environment.

NLS_ISO_CURRENCY

Parameter type	String
Syntax	NLS_ISO_CURRENCY = *territory*
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic : ALTER SESSION
Range of values	Any valid NLS_TERRITORY value

NLS_ISO_CURRENCY specifies the string to use as the international currency symbol for the C number format element.

NLS_LANGUAGE

Parameter type	String
Syntax	NLS_LANGUAGE = *language*
Default value	Operating system-dependent, derived from the NLS_LANG environment variable
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid language name

NLS_LANGUAGE specifies the default language of the database. This language is used for messages, day and month names, symbols for AD, BC, a.m., and p.m., and the default sorting mechanism. This parameter also determines the default values of the parameters NLS_DATE_LANGUAGE and NLS_SORT.

NLS_LENGTH_SEMANTICS

Parameter type	String
Syntax	NLS_LENGTH_SEMANTICS = *string* Example: NLS_LENGTH_SEMANTICS = 'CHAR'
Default value	BYTE
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM
Range of values	BYTE | CHAR

NLS_LENGTH_SEMANTICS enables you to create CHAR and VARCHAR2 columns using either byte or character length semantics. Existing columns are not affected.

NLS_NCHAR_CONV_EXCP

Parameter type	String
Syntax	NLS_NCHAR_CONV_EXCP = {TRUE | FALSE}
Default value	FALSE
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

NLS_NCHAR_CONV_EXCP determines whether data loss during an implicit or explicit character type conversion will report an error.

NLS_NUMERIC_CHARACTERS

Parameter type	String
Syntax	NLS_NUMERIC_CHARACTERS = "*decimal_character group_separator*"
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic: ALTER SESSION

NLS_NUMERIC_CHARACTERS specifies the characters to use as the group separator and decimal character. It overrides those characters defined implicitly by NLS_TERRITORY. The group separator separates integer groups (that is, thousands, millions, billions, and so on). The decimal separates the integer portion of a number from the decimal portion.

NLS_SORT

Parameter type	String
Syntax	NLS_SORT = {BINARY | *linguistic_definition*}
Default value	Derived from NLS_LANGUAGE
Parameter class	Dynamic: ALTER SESSION
Range of values	BINARY or any valid linguistic definition name

NLS_SORT specifies the collating sequence for ORDER BY queries.

NLS_TERRITORY

Parameter type	String
Syntax	NLS_TERRITORY = *territory*
Default value	Operating system-dependent
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid territory name

NLS_TERRITORY specifies the name of the territory whose conventions are to be followed for day and week numbering.

NLS_TIMESTAMP_FORMAT

Parameter type	String
Syntax	NLS_TIMESTAMP_FORMAT = "*format*"
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid datetime format mask

NLS_TIMESTAMP_FORMAT defines the default timestamp format to use with the TO_CHAR and TO_TIMESTAMP functions.

NLS_TIMESTAMP_TZ_FORMAT

Parameter type	String
Syntax	NLS_TIMESTAMP_TZ_FORMAT = "*format*"
Default value	Derived from NLS_TERRITORY
Parameter class	Dynamic: ALTER SESSION
Range of values	Any valid datetime format mask

NLS_TIMESTAMP_TZ_FORMAT defines the default timestamp with time zone format to use with the TO_CHAR and TO_TIMESTAMP_TZfunctions.

O7_DICTIONARY_ACCESSIBILITY

Parameter type	Boolean
Default value	false
Parameter class	Static
Range of values	true | false

O7_DICTIONARY_ACCESSIBILITY is intended for use when you migrate from Oracle7 to Oracle Security Server. It controls restrictions on SYSTEM privileges. If the parameter is set to true, access to objects in the SYS schema is allowed (Oracle7

behavior). The default setting of `false` ensures that system privileges that allow access to objects in "any schema" do not allow access to objects in SYS schema.

OBJECT_CACHE_MAX_SIZE_PERCENT

Parameter type Integer

Default value 10

Parameter class Dynamic: `ALTER SESSION`, `ALTER SYSTEM ... DEFERRED`

Range of values 0 to operating system-dependent maximum

The **object cache** is a memory block on the client that allows applications to store entire objects and to navigate among them without round trips to the server. `OBJECT_CACHE_MAX_SIZE_PERCENT` specifies the percentage of the optimal cache size that the session object cache can grow past the optimal size. The maximum size is equal to the optimal size plus the product of this percentage and the optimal size. When the cache size exceeds this maximum size, the system will attempt to shrink the cache to the optimal size.

OBJECT_CACHE_OPTIMAL_SIZE

Parameter type Integer

Default value `102400` (100K)

Parameter class Dynamic: `ALTER SESSION`, `ALTER SYSTEM ... DEFERRED`

Range of values 10 KB to operating system-dependent maximum

The **object cache** is a memory block on the client that allows applications to store entire objects and to navigate among them without round trips to the server. `OBJECT_CACHE_OPTIMAL_SIZE` specifies (in bytes) the size to which the session object cache is reduced when the size of the cache exceeds the maximum size.

OLAP_PAGE_POOL_SIZE

Parameter type Integer

Default value 32 MB

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM ... DEFERRED

Range of values 32 MB to 2 GB

OLAP_PAGE_POOL_SIZE specifies the size (in bytes) of the OLAP pool.

OPEN_CURSORS

Parameter type Integer

Default value 50

Parameter class Static

Range of values 1 to 4294967295 (4 GB -1)

OPEN_CURSORS specifies the maximum number of open cursors (handles to private SQL areas) a session can have at once. You can use this parameter to prevent a session from opening an excessive number of cursors. This parameter also constrains the size of the PL/SQL cursor cache which PL/SQL uses to avoid having to reparse as statements are reexecuted by a user.

OPEN_LINKS

Parameter type Integer

Default value 4

Parameter class Static

Range of values 0 to 255

OPEN_LINKS specifies the maximum number of concurrent open connections to remote databases in one session. These connections include database links, as well as external procedures and cartridges, each of which uses a separate process.

OPEN_LINKS_PER_INSTANCE

Parameter type Integer

Default value 4

Parameter class Static

Range of values 0 to 4294967295 (4 GB -1)

Real Application Clusters Multiple instances can have different values.

OPEN_LINKS_PER_INSTANCE specifies the maximum number of migratable open connections globally for each database instance. XA transactions use migratable open connections so that the connections are cached after a transaction is committed. Another transaction can use the connection, provided the user who created the connection is the same as the user who owns the transaction.

OPTIMIZER_DYNAMIC_SAMPLING

Parameter type Integer

Default value If OPTIMIZER_FEATURES_ENABLE is set to 9.2.0 or higher, then 1

If OPTIMIZER_FEATURES_ENABLE is set to 9.0.1 or lower, then 0

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM

Range of values 0 to 10

OPTIMIZER_DYNAMIC_SAMPLING controls the level of dynamic sampling performed by the optimizer.

OPTIMIZER_FEATURES_ENABLE

Parameter type String

Syntax
```
OPTIMIZER_FEATURES_ENABLE = {8.0.0 | 8.0.3 |
8.0.4 | 8.0.5 | 8.0.6 | 8.0.7 | 8.1.0 | 8.1.3
| 8.1.4 | 8.1.5 | 8.1.6 | 8.1.7 | 9.0.0 |
9.0.1 | 9.2.0}
```

Default value 9.2.0

Parameter class Static

OPTIMIZER_FEATURES_ENABLE acts as an umbrella parameter for enabling a series of optimizer features based on an Oracle release number.

OPTIMIZER_INDEX_CACHING

Parameter type Integer

Default value	0
Parameter class	Dynamic: ALTER SESSION
Range of values	0 to 100

OPTIMIZER_INDEX_CACHING lets you adjust the behavior of cost-based optimization to favor nested loops joins and IN-list iterators.

OPTIMIZER_INDEX_COST_ADJ

Parameter type	Integer
Default value	100
Parameter class	Dynamic: ALTER SESSION
Range of values	1 to 10000

OPTIMIZER_INDEX_COST_ADJ lets you tune optimizer behavior for access path selection to be more or less index friendly—that is, to make the optimizer more or less prone to selecting an index access path over a full table scan.

OPTIMIZER_MAX_PERMUTATIONS

Parameter type	Integer
Default value	If OPTIMIZER_FEATURES_ENABLE is set to 9.0.0 or higher, then 2000 If OPTIMIZER_FEATURES_ENABLE is set to 8.1.7 or lower, then 80000
Parameter class	Dynamic: ALTER SESSION
Range of values	4 to 80000

OPTIMIZER_MAX_PERMUTATIONS restricts the number of permutations of the tables the optimizer will consider in queries with joins. Such a restriction ensures that the parse time for the query stays within acceptable limits. However, a slight risk exists that the optimizer will overlook a good plan it would otherwise have found.

OPTIMIZER_MODE

Parameter type	String
Syntax	OPTIMIZER_MODE = {first_rows_[1 | 10 | 100 | 1000] | first_rows | all_rows | choose | rule}
Default value	choose
Parameter class	Dynamic: ALTER SESSION

OPTIMIZER_MODE establishes the default behavior for choosing an optimization approach for the instance.

ORACLE_TRACE_COLLECTION_NAME

Parameter type	String
Syntax	ORACLE_TRACE_COLLECTION_NAME = *collection_name*
Default value	There is no default value.
Parameter class	Static
Range of values	Valid collection name up to 16 characters long (except for platforms that enforce 8-character file names)

A **collection** is data collected for events that occurred while an instrumented product was running. ORACLE_TRACE_COLLECTION_NAME specifies the Oracle Trace collection name for this instance. Oracle also uses this parameter in the output file names (collection definition file .cdf and data collection file .dat). If you set ORACLE_TRACE_ENABLE to true, setting this value to a non-null string will start a default Oracle Trace collection that will run until this value is set to null again.

ORACLE_TRACE_COLLECTION_PATH

Parameter type	String
Syntax	ORACLE_TRACE_COLLECTION_PATH = *pathname*
Default value	Operating system-specific
Parameter class	Static

Range of values Full directory pathname

ORACLE_TRACE_COLLECTION_PATH specifies the directory pathname where the Oracle Trace collection definition (.cdf) and data collection (.dat) files are located. If you accept the default, the Oracle Trace .cdf and .dat files will be located in *ORACLE_HOME*/otrace/admin/cdf.

ORACLE_TRACE_COLLECTION_SIZE

Parameter type Integer

Default value 5242880

Parameter class Static

Range of values 0 to 4294967295

ORACLE_TRACE_COLLECTION_SIZE specifies (in bytes) the maximum size of the Oracle Trace collection file (.dat). Once the collection file reaches this maximum, the collection is disabled. A value of 0 means that the file has no size limit.

ORACLE_TRACE_ENABLE

Parameter type Boolean

Default value false

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM

Range of values true | false

To enable Oracle Trace collections for the server, set ORACLE_TRACE_ENABLE to true. This setting alone does not start an Oracle Trace collection, but it allows Oracle Trace to be used for the server.

ORACLE_TRACE_FACILITY_NAME

Parameter type String

Syntax
```
ORACLE_TRACE_FACILITY_NAME =
   {ORACLED | ORACLEE | ORACLESM | ORACLEC}
```

Default value ORACLED

Parameter class Static

ORACLE_TRACE_FACILITY_NAME specifies the event set that Oracle Trace collects. The value of this parameter, followed by the .fdf extension, is the name of the Oracle Trace product definition file. That file must be located in the directory specified by the ORACLE_TRACE_FACILITY_PATH parameter. The product definition file contains definition information for all the events and data items that can be collected for products that use the Oracle Trace data collection API.

ORACLE_TRACE_FACILITY_PATH

Parameter type	String
Syntax	ORACLE_TRACE_FACILITY_PATH = *pathname*
Default value	Operating system-specific
Parameter class	Static
Range of values	Full directory pathname

ORACLE_TRACE_FACILITY_PATH specifies the directory pathname where Oracle Trace facility definition files are located. On Solaris, the default path is *ORACLE_HOME*/otrace/admin/fdf/. On NT, the default path is %OTRACE80%\ADMIN\FDF\.

OS_AUTHENT_PREFIX

Parameter type	String
Syntax	OS_AUTHENT_PREFIX = *authentication_prefix*
Default value	OPS$
Parameter class	Static

OS_AUTHENT_PREFIX specifies a prefix that Oracle uses to authenticate users attempting to connect to the server. Oracle concatenates the value of this parameter to the beginning of the user's operating system account name and password. When a connection request is attempted, Oracle compares the prefixed username with Oracle usernames in the database.

OS_ROLES

Parameter type	Boolean
Default value	false

Parameter class Static

Range of values `true | false`

`OS_ROLES` determines whether Oracle or the operating system identifies and manages the roles of each username.

PARALLEL_ADAPTIVE_MULTI_USER

Parameter type Boolean

Default value Derived from the value of `PARALLEL_AUTOMATIC_TUNING`

Parameter class Dynamic: `ALTER SYSTEM`

Range of values `true | false`

`PARALLEL_ADAPTIVE_MULTI_USER`, when set to `true`, enables an adaptive algorithm designed to improve performance in multiuser environments that use parallel execution. The algorithm automatically reduces the requested degree of parallelism based on the system load at query startup time. The effective degree of parallelism is based on the default degree of parallelism, or the degree from the table or hints, divided by a reduction factor.

PARALLEL_AUTOMATIC_TUNING

Parameter type Boolean

Default value `false`

Parameter class Static

Range of values `true | false`

> **Note:** This parameter applies to parallel execution in exclusive mode as well as in the Oracle9*i* Real Application Clusters environment.

When `PARALLEL_AUTOMATIC_TUNING` is set to `true`, Oracle determines the default values for parameters that control parallel execution. In addition to setting this parameter, you must specify the `PARALLEL` clause for the target tables in the system. Oracle then tunes all subsequent parallel operations automatically.

PARALLEL_EXECUTION_MESSAGE_SIZE

Parameter type	Integer
Default value	Operating system-dependent
Parameter class	Static
Range of values	2148 to 65535 (64 KB - 1)
Real Application Clusters	Multiple instances must have the same value.

PARALLEL_EXECUTION_MESSAGE_SIZE specifies the size of messages for parallel execution (formerly referred to as parallel query, PDML, Parallel Recovery, replication).

PARALLEL_INSTANCE_GROUP

Parameter type	String
Syntax	PARALLEL_INSTANCE_GROUP = *group_name*
Default value	A group consisting of all instances currently active
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM
Range of values	Any group name specified in the INSTANCE_GROUPS parameter of any active instance
Real Application Clusters	Different instances can have different values.

PARALLEL_INSTANCE_GROUP is an Oracle9*i* Real Application Clusters parameter that you can specify in parallel mode only. Used in conjunction with the INSTANCE_GROUPS parameter, it lets you restrict parallel query operations to a limited number of instances.

PARALLEL_MAX_SERVERS

Parameter type	Integer
Default value	Derived from the values of CPU_COUNT, PARALLEL_AUTOMATIC_TUNING, and PARALLEL_ADAPTIVE_MULTI_USER
Parameter class	Static

Range of values	0 to 3599
Real Application Clusters	Multiple instances must have the same value.

> **Note:** This parameter applies to parallel execution in exclusive mode as well as in the Oracle9*i* Real Application Clusters environment.

PARALLEL_MAX_SERVERS specifies the maximum number of parallel execution processes and parallel recovery processes for an instance. As demand increases, Oracle increases the number of processes from the number created at instance startup up to this value.

PARALLEL_MIN_PERCENT

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SESSION
Range of values	0 to 100
Real Application Clusters	Multiple instances can have different values.

PARALLEL_MIN_PERCENT operates in conjunction with PARALLEL_MAX_SERVERS and PARALLEL_MIN_SERVERS. It lets you specify the minimum percentage of parallel execution processes (of the value of PARALLEL_MAX_SERVERS) required for parallel execution. Setting this parameter ensures that parallel operations will not execute sequentially unless adequate resources are available. The default value of 0 means that no minimum percentage of processes has been set.

PARALLEL_MIN_SERVERS

Parameter type	Integer
Default value	0
Parameter class	Static
Range of values	0 to value of PARALLEL_MAX_SERVERS

Real Application Clusters Multiple instances can have different values.

> **Note:** This parameter applies to parallel execution in exclusive mode as well as in the Oracle9*i* Real Application Clusters environment.

PARALLEL_MIN_SERVERS specifies the minimum number of parallel execution processes for the instance. This value is the number of parallel execution processes Oracle creates when the instance is started.

PARALLEL_THREADS_PER_CPU

Parameter type Integer

Default value Operating system-dependent, usually 2

Parameter class Dynamic: ALTER SYSTEM

Range of values Any nonzero number

> **Note:** This parameter applies to parallel execution in exclusive mode as well as in the Oracle9*i* Real Application Clusters environment.

PARALLEL_THREADS_PER_CPU specifies the default degree of parallelism for the instance and determines the parallel adaptive and load balancing algorithms. The parameter describes the number of parallel execution processes or **threads** that a CPU can handle during parallel execution.

PARTITION_VIEW_ENABLED

Parameter type Boolean

Default value false

Parameter class Dynamic: ALTER SESSION

Range of values true | false

Note: Oracle Corporation recommends that you use partitioned tables (available starting with Oracle8) rather than partition views. Partition views are supported for backward compatibility only.

PARTITION_VIEW_ENABLED specifies whether the optimizer uses partition views. If you set this parameter to `true`, the optimizer prunes (or skips) unnecessary table accesses in a partition view and alters the way it computes statistics on a partition view from statistics on underlying tables.

PGA_AGGREGATE_TARGET

Parameter type	Big integer
Syntax	PGA_AGGREGATE_TARGET = *integer* [K | M | G]
Default value	0 (automatic memory management is turned OFF by default)
Parameter class	Dynamic: ALTER SYSTEM
Range of values	10 MB to 4000 GB

PGA_AGGREGATE_TARGET specifies the target aggregate PGA memory available to all server processes attached to the instance. You must set this parameter to enable the automatic sizing of SQL working areas used by memory-intensive SQL operators such as sort, group-by, hash-join, bitmap merge, and bitmap create.

PLSQL_COMPILER_FLAGS

Parameter type	String
Syntax	PLSQL_COMPILER_FLAGS = { [DEBUG | NON_DEBUG] [INTERPRETED | NORMAL] }
Default value	INTERPRETED, NON_DEBUG
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

PLSQL_COMPILER_FLAGS is a parameter used by the PL/SQL compiler. It specifies a list of compiler flags as a comma-separated list of strings.

PLSQL_NATIVE_C_COMPILER

Parameter type	String
Syntax	PLSQL_NATIVE_C_COMPILER = *pathname*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid path name

PLSQL_NATIVE_C_COMPILER specifies the full path name of a C compiler which is used to compile the generated C file into an object file.

PLSQL_NATIVE_LIBRARY_DIR

Parameter type	String
Syntax	PLSQL_NATIVE_LIBRARY_DIR = *directory*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid directory path

PLSQL_NATIVE_LIBRARY_DIR is a parameter used by the PL/SQL compiler. It specifies the name of a directory where the shared objects produced by the native compiler are stored.

PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to 2^{32}-1 (max value represented by 32 bits)

PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT specifies the number of subdirectories created by the database administrator in the directory specified by PLSQL_NATIVE_LIBRARY_DIR.

PLSQL_NATIVE_LINKER

Parameter type	String
Syntax	PLSQL_NATIVE_LINKER = *pathname*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid path name

PLSQL_NATIVE_LINKER specifies the full path name of a linker such as ld in UNIX or GNU ld which is used to link the object file into a shared object or DLL.

PLSQL_NATIVE_MAKE_FILE_NAME

Parameter type	String
Syntax	PLSQL_NATIVE_MAKE_FILE_NAME = *pathname*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid path name

PLSQL_NATIVE_MAKE_FILE_NAME specifies the full path name of a make file. The make utility (specified by *PLSQL_NATIVE_MAKE_UTILITY*) uses this make file to generate the shared object or DLL.

PLSQL_NATIVE_MAKE_UTILITY

Parameter type	String
Syntax	PLSQL_NATIVE_MAKE_UTILITY = *pathname*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid path name

PLSQL_NATIVE_MAKE_UTILITY specifies the full path name of a make utility such as make in UNIX or gmake (GNU make). The make utility is needed to generate the shared object or DLL from the generated C source.

PLSQL_V2_COMPATIBILITY

Parameter type	Boolean
Default value	`false`
Parameter class	Dynamic: `ALTER SESSION`, `ALTER SYSTEM`
Range of values	`true | false`

PL/SQL Version 2 allows some abnormal behavior that Version 8 disallows. If you want to retain that behavior for backward compatibility, set `PLSQL_V2_COMPATIBILITY` to `true`. If you set it to `false`, PL/SQL Version 8 behavior is enforced and Version 2 behavior is not allowed.

PRE_PAGE_SGA

Parameter type	Boolean
Default value	`false`
Parameter class	Static
Range of values	`true | false`

`PRE_PAGE_SGA` determines whether Oracle reads the entire SGA into memory at instance startup. Operating system page table entries are then prebuilt for each page of the SGA. This setting can increase the amount of time necessary for instance startup, but it is likely to decrease the amount of time necessary for Oracle to reach its full performance capacity after startup.

PROCESSES

Parameter type	Integer
Default value	Derived from `PARALLEL_MAX_SERVERS`
Parameter class	Static
Range of values	6 to operating system-dependent
Real Application Clusters	Multiple instances can have different values.

`PROCESSES` specifies the maximum number of operating system user processes that can simultaneously connect to Oracle. Its value should allow for all background processes such as locks, job queue processes, and parallel execution processes.

QUERY_REWRITE_ENABLED

Parameter type	String
Syntax	`QUERY_REWRITE_ENABLED = {force | true | false}`
Default value	`false`
Parameter class	Dynamic: `ALTER SESSION`, `ALTER SYSTEM`
Real Application Clusters	Multiple instances can have different values.

`QUERY_REWRITE_ENABLED` allows you to enable or disable query rewriting globally for the database.

See Also: "Enabling Query Rewrite: Example" on page 10-122

QUERY_REWRITE_INTEGRITY

Parameter type	String
Syntax	`QUERY_REWRITE_INTEGRITY = {stale_tolerated | trusted | enforced}`
Default value	`enforced`
Parameter class	Dynamic: `ALTER SESSION`, `ALTER SYSTEM`
Real Application Clusters	Multiple instances can have different values.

`QUERY_REWRITE_INTEGRITY` determines the degree to which Oracle must enforce query rewriting. At the safest level, Oracle does not use query rewrite transformations that rely on unenforced relationships.

RDBMS_SERVER_DN

Parameter type	X.500 Distinguished Name
Default value	There is no default value.
Parameter class	Static
Range of values	All X.500 Distinguished Name format values

`RDBMS_SERVER_DN` specifies the Distinguished Name (DN) of the Oracle server. It is used for retrieving Enterprise Roles from an enterprise directory service.

READ_ONLY_OPEN_DELAYED

Parameter type Boolean

Default value `false`

Parameter class Static

Range of values `true | false`

`READ_ONLY_OPEN_DELAYED` determines when datafiles in read-only tablespaces are accessed.

RECOVERY_PARALLELISM

Parameter type Integer

Default value Operating system-dependent

Parameter class Static

Range of values Operating system-dependent, but cannot exceed `PARALLEL_MAX_SERVERS`

`RECOVERY_PARALLELISM` specifies the number of processes to participate in instance or crash recovery. A value of `0` or `1` indicates that recovery is to be performed serially by one process.

REMOTE_ARCHIVE_ENABLE

Parameter type String

Syntax
```
REMOTE_ARCHIVE_ENABLE = {receive [, send] |
false | true}
```

Default value `true`

Parameter class Static

`REMOTE_ARCHIVE_ENABLE` enables or disables the sending of redo archival to remote destinations and the receipt of remotely archived redo.

REMOTE_DEPENDENCIES_MODE

Parameter type String

Syntax	REMOTE_DEPENDENCIES_MODE = {TIMESTAMP | SIGNATURE}
Default value	TIMESTAMP
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

REMOTE_DEPENDENCIES_MODE specifies how Oracle should handle dependencies upon remote PL/SQL stored procedures.

REMOTE_LISTENER

Parameter type	String
Syntax	REMOTE_LISTENER = *network_name*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM

REMOTE_LISTENER specifies a network name that resolves to an address or address list of Oracle Net remote listeners (that is, listeners that are not running on the same machine as this instance). The address or address list is specified in the TNSNAMES.ORA file or other address repository as configured for your system.

REMOTE_LOGIN_PASSWORDFILE

Parameter type	String
Syntax	REMOTE_LOGIN_PASSWORDFILE= {NONE | SHARED | EXCLUSIVE}
Default value	NONE
Parameter class	Static
Real Application Clusters	Multiple instances must have the same value.

REMOTE_LOGIN_PASSWORDFILE specifies whether Oracle checks for a password file and how many databases can use the password file.

REMOTE_OS_AUTHENT

Parameter type	Boolean

Default value	`false`
Parameter class	Static
Range of values	`true | false`

`REMOTE_OS_AUTHENT` specifies whether remote clients will be authenticated with the value of the `OS_AUTHENT_PREFIX` parameter.

REMOTE_OS_ROLES

Parameter type	Boolean
Default value	`false`
Parameter class	Static
Range of values	`true | false`

`REMOTE_OS_ROLES` specifies whether operating system roles are allowed for remote clients. The default value, `false`, causes Oracle to identify and manage roles for remote clients.

REPLICATION_DEPENDENCY_TRACKING

Parameter type	Boolean
Default value	`true`
Parameter class	Static
Range of values	`true | false`

`REPLICATION_DEPENDENCY_TRACKING` enables or disables dependency tracking for read / write operations to the database. Dependency tracking is essential for propagating changes in a replicated environment in parallel.

RESOURCE_LIMIT

Parameter type	Boolean
Default value	`false`
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	`true | false`

RESOURCE_LIMIT determines whether resource limits are enforced in database profiles.

See Also: "Enabling Resource Limits: Example" on page 10-122

RESOURCE_MANAGER_PLAN

Parameter type	String
Syntax	RESOURCE_MANAGER_PLAN = *plan_name*
Default value	There is no default value.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid character string

RESOURCE_MANAGER_PLAN specifies the top-level resource plan to use for an instance. The resource manager will load this top-level plan along with all its descendants (subplans, directives, and consumer groups). If you do not specify this parameter, the resource manager is off by default.

ROLLBACK_SEGMENTS

Parameter type	String
Syntax	ROLLBACK_SEGMENTS = (segment_name [, segment_name] ...)
Default value	The instance uses public rollback segments by default if you do not specify this parameter
Parameter class	Static
Range of values	Any rollback segment names listed in DBA_ROLLBACK_SEGS except SYSTEM
Real Application Clusters	Multiple instances must have different values.

ROLLBACK_SEGMENTS allocates one or more rollback segments by name to this instance. If you set this parameter, the instance acquires all of the rollback segments named in this parameter, even if the number of rollback segments exceeds the minimum number required by the instance (calculated as TRANSACTIONS / TRANSACTIONS_PER_ROLLBACK_SEGMENT).

ROW_LOCKING

Parameter type	String
Syntax	ROW_LOCKING = {ALWAYS | DEFAULT | INTENT}
Default value	ALWAYS
Parameter class	Static
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have the same value.

ROW_LOCKING specifies whether row locks are acquired during UPDATE operations.

SERIAL_REUSE

Parameter type	String
Syntax	SERIAL_REUSE = {DISABLE | SELECT | DML | PLSQL | ALL}
Default value	DISABLE
Parameter class	Static

SERIAL_REUSE specifies which types of cursors make use of the serial-reusable memory feature. This feature allocates private cursor memory in the SGA so that it can be reused (serially, not concurrently) by sessions executing the same cursor.

SERVICE_NAMES

Parameter type	String
Syntax	SERVICE_NAMES = *db_service_name* [, *db_service_name* [,...]]
Default value	DB_NAME.DB_DOMAIN if defined
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any ASCII string or comma-separated list of string names

SERVICE_NAMES specifies one or more names for the database service to which this instance connects. You can specify multiple service names in order to distinguish among different uses of the same database.

SESSION_CACHED_CURSORS

Parameter type	Integer
Default value	0
Parameter class	Dynamic: ALTER SESSION
Range of values	0 to operating system-dependent
Real Application Clusters	Multiple instances can have different values.

SESSION_CACHED_CURSORS lets you specify the number of session cursors to cache. Repeated parse calls of the same SQL statement cause the session cursor for that statement to be moved into the session cursor cache. Subsequent parse calls will find the cursor in the cache and do not need to reopen the cursor. Oracle uses a least recently used algorithm to remove entries in the session cursor cache to make room for new entries when needed.

SESSION_MAX_OPEN_FILES

Parameter type	Integer
Default value	10
Parameter class	Static
Range of values	1 to either 50 or the value of MAX_OPEN_FILES defined at the operating system level, whichever is less

SESSION_MAX_OPEN_FILES specifies the maximum number of BFILEs that can be opened in any session. Once this number is reached, subsequent attempts to open more files in the session by using DBMS_LOB.FILEOPEN() or OCILobFileOpen() will fail. The maximum value for this parameter depends on the equivalent parameter defined for the underlying operating system.

SESSIONS

Parameter type	Integer

Default value	Derived: (1.1 * PROCESSES) + 5
Parameter class	Static
Range of values	1 to 2^{31}

SESSIONS specifies the maximum number of sessions that can be created in the system. Because every login requires a session, this parameter effectively determines the maximum number of concurrent users in the system. You should always set this parameter explicitly to a value equivalent to your estimate of the maximum number of concurrent users, plus the number of background processes, plus approximately 10% for recursive sessions.

SGA_MAX_SIZE

Parameter type	Big integer
Syntax	SGA_MAX_SIZE = *integer* [K | M | G]
Default value	Initial size of SGA at startup, dependent on the sizes of different pools in the SGA, such as buffer cache, shared pool, large pool, and so on.
Parameter class	Static
Range of values	0 to operating system-dependent

SGA_MAX_SIZE specifies the maximum size of SGA for the lifetime of the instance.

SHADOW_CORE_DUMP

Parameter type	String
Syntax	SHADOW_CORE_DUMP = {partial | full | none}
Default value	partial
Parameter class	Static

SHADOW_CORE_DUMP specifies whether Oracle includes the SGA in the core file for foreground (client) processes.

SHARED_MEMORY_ADDRESS

Parameter type	Integer

Default value	0
Parameter class	Static

SHARED_MEMORY_ADDRESS and HI_SHARED_MEMORY_ADDRESS specify the starting address at runtime of the system global area (SGA). This parameter is ignored on the many platforms that specify the SGA's starting address at linktime.

SHARED_POOL_RESERVED_SIZE

Parameter type	Big integer
Syntax	SHARED_POOL_RESERVED_SIZE = *integer* [K | M | G]
Default value	5% of the value of SHARED_POOL_SIZE
Parameter class	Static
Range of values	Minimum: 5000 Maximum: one half of the value of SHARED_POOL_SIZE

SHARED_POOL_RESERVED_SIZE specifies (in bytes) the shared pool space that is reserved for large contiguous requests for shared pool memory.

SHARED_POOL_SIZE

Parameter type	Big integer
Syntax	SHARED_POOL_SIZE = *integer* [K | M | G]
Default value	32-bit platforms: 8 MB, rounded up to the nearest granule size 64-bit platforms: 64 MB, rounded up to the nearest granule size
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Minimum: the granule size Maximum: operating system-dependent

SHARED_POOL_SIZE specifies (in bytes) the size of the shared pool. The shared pool contains shared cursors, stored procedures, control structures, and other structures. If you set PARALLEL_AUTOMATIC_TUNING to false, then Oracle also

allocates parallel execution message buffers from the shared pool. Larger values improve performance in multi-user systems. Smaller values use less memory.

Shared Server Parameters

Beginning in Oracle9*i*, the multi-threaded server architecture is called **shared server architecture**.

When you start your instance, Oracle creates shared server processes and dispatcher processes for the shared server architecture based on the values of the SHARED_SERVERS and DISPATCHERS initialization parameters. You can also set the SHARED_SERVERS and DISPATCHERS parameters with ALTER SYSTEM to perform one of the following operations while the instance is running:

- Create additional shared server processes by increasing the minimum number of shared server processes.
- Terminate existing shared server processes after their current calls finish processing.
- Create more dispatcher processes for a specific protocol, up to a maximum across all protocols specified by the initialization parameter MAX_DISPATCHERS.
- Terminate existing dispatcher processes for a specific protocol after their current user processes disconnect from the instance.

SHARED_SERVERS

Parameter type	Integer
Default value	If you are using shared server architecture, then the value is 1. If you are not using shared server architecture, then the value is 0.
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Operating system-dependent

SHARED_SERVERS specifies the number of server processes that you want to create when an instance is started up. If system load decreases, this minimum number of servers is maintained. Therefore, you should take care not to set SHARED_SERVERS too high at system startup.

See Also: "Changing Shared Server Settings: Examples" on page 10-122

SHARED_SERVER_SESSIONS

Parameter type	Integer
Default value	Derived: the lesser of CIRCUITS and SESSIONS - 5
Parameter class	Static
Range of values	0 to SESSIONS - 5

SHARED_SERVER_SESSIONS specifies the total number of shared server architecture user sessions to allow. Setting this parameter enables you to reserve user sessions for dedicated servers.

SORT_AREA_RETAINED_SIZE

Parameter type	Integer
Default value	Derived from SORT_AREA_SIZE
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM ... DEFERRED
Range of values	From the value equivalent of two database blocks to the value of SORT_AREA_SIZE

Note: Oracle does not recommend using the SORT_AREA_RETAINED_SIZE parameter unless the instance is configured with the shared server option. Oracle recommends that you enable automatic sizing of SQL working areas by setting PGA_AGGREGATE_TARGET instead. SORT_AREA_RETAINED_SIZE is retained for backward compatibility.

SORT_AREA_RETAINED_SIZE specifies (in bytes) the maximum amount of the user global area (UGA) memory retained after a sort run completes. The retained size controls the size of the read buffer, which Oracle uses to maintain a portion of the sort in memory. This memory is released back to the UGA, not to the operating system, after the last row is fetched from the sort space.

SORT_AREA_SIZE

Parameter type	Integer
Default value	65536
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM ... DEFERRED
Range of values	Minimum: the value equivalent of six database blocks Maximum: operating system-dependent

> **Note:** Oracle does not recommend using the SORT_AREA_SIZE parameter unless the instance is configured with the shared server option. Oracle recommends that you enable automatic sizing of SQL working areas by setting PGA_AGGREGATE_TARGET instead. SORT_AREA_SIZE is retained for backward compatibility.

SORT_AREA_SIZE specifies in bytes the maximum amount of memory Oracle will use for a sort. After the sort is complete, but before the rows are returned, Oracle releases all of the memory allocated for the sort, except the amount specified by the SORT_AREA_RETAINED_SIZE parameter. After the last row is returned, Oracle releases the remainder of the memory.

SPFILE

Parameter type	String
Syntax	SPFILE = *spfile_name*
Default value	*ORACLE_HOME*/dbs/spfile.ora
Parameter class	Static
Range of values	Any valid SPFILE
Real Application Clusters	Multiple instances should have the same value.

The value of this parameter is the name of the current server parameter file (SPFILE) in use. This parameter can be defined in a client side PFILE to indicate the name of the server parameter file to use.

SQL92_SECURITY

Parameter type Boolean

Default value `false`

Parameter class Static

Range of values `true | false`

The SQL92 standards specify that security administrators should be able to require that users have `SELECT` privilege on a table when executing an `UPDATE` or `DELETE` statement that references table column values in a `WHERE` or `SET` clause. `SQL92_SECURITY` lets you specify whether users must have been granted the `SELECT` object privilege in order to execute such `UPDATE` or `DELETE` statements.

SQL_TRACE

Parameter type Boolean

Default value `false`

Parameter class Static

Range of values `true | false`

The value of `SQL_TRACE` disables or enables the SQL trace facility. Setting this parameter to `true` provides information on tuning that you can use to improve performance. You can change the value using the `DBMS_SYSTEM` package.

STANDBY_ARCHIVE_DEST

Parameter type String

Syntax `STANDBY_ARCHIVE_DEST = filespec`

Default value Operating system-specific

Parameter class Dynamic: `ALTER SYSTEM`

Range of values A valid path or device name other than `RAW`

`STANDBY_ARCHIVE_DEST` is relevant only for a standby database in managed recovery mode. It specifies the location of archive logs arriving from a primary database. Oracle uses `STANDBY_ARCHIVE_DEST` and `LOG_ARCHIVE_FORMAT` to fabricate the fully qualified standby log filenames and stores the filenames in the standby control file.

STANDBY_FILE_MANAGEMENT

Parameter type String

Syntax STANDBY_FILE_MANAGEMENT = {MANUAL | AUTO}

Default value MANUAL

Parameter class Dynamic: ALTER SYSTEM

STANDBY_FILE_MANAGEMENT enables or disables automatic standby file management. When automatic standby file management is enabled, operating system file additions and deletions on the primary database are replicated on the standby database.

STAR_TRANSFORMATION_ENABLED

Parameter type String

Syntax STAR_TRANSFORMATION_ENABLED =
{TEMP_DISABLE | TRUE | FALSE}

Default value FALSE

Parameter class Dynamic: ALTER SESSION

STAR_TRANSFORMATION_ENABLED determines whether a cost-based query transformation will be applied to star queries.

STATISTICS_LEVEL

Parameter type String

Syntax STATISTICS_LEVEL = {ALL | TYPICAL | BASIC}

Default value TYPICAL

Parameter class Dynamic: ALTER SESSION, ALTER SYSTEM

STATISTICS_LEVEL sets the statistics collection level of the database.

TAPE_ASYNCH_IO

Parameter type Boolean

Default value true

Parameter class	Static
Range of values	`true | false`

TAPE_ASYNCH_IO controls whether I/O to sequential devices (for example, backup or restore of Oracle data to or from tape) is asynchronous—that is, whether parallel server processes can overlap I/O requests with CPU processing during table scans. If your platform supports asynchronous I/O to sequential devices, Oracle Corporation recommends that you leave this parameter set to its default. However, if the asynchronous I/O implementation is not stable, you can set TAPE_ASYNCH_IO to `false` to disable asynchronous I/O. If your platform does not support asynchronous I/O to sequential devices, this parameter has no effect.

THREAD

Parameter type	Integer
Default value	`0`
Parameter class	Static
Range of values	0 to the maximum number of enabled threads
Real Application Clusters	If specified, multiple instances must have different values.

THREAD is an Oracle9*i* Real Application Clusters parameter that specifies the number of the redo thread to be used by an instance.

TIMED_OS_STATISTICS

Parameter type	Integer
Default value	If STATISTICS_LEVEL is set to ALL, then 5
	If STATISTICS_LEVEL is set to BASIC or TYPICAL, then 0
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM
Range of values	Unlimited

TIMED_OS_STATISTICS specifies the interval (in seconds) at which Oracle collects operating system statistics when a request is made from the client to the server or when a request completes.

TIMED_STATISTICS

Parameter type	Boolean
Default value	If STATISTICS_LEVEL is set to TYPICAL or ALL, then true If STATISTICS_LEVEL is set to BASIC, then false
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM
Range of values	true | false

TIMED_STATISTICS specifies whether or not statistics related to time are collected.

TRACE_ENABLED

Parameter type	Boolean
Default value	true
Parameter class	Dynamic: ALTER SYSTEM
Range of values	true | false
Real Application Clusters	You must set this parameter for every instance, and multiple instances must have the same value.

TRACE_ENABLED controls tracing of the execution history, or code path, of Oracle. Oracle Support Services uses this information for debugging.

TRACEFILE_IDENTIFIER

Parameter type	String
Syntax	TRACEFILE_IDENTIFIER = "*traceid*"
Default value	There is no default value.
Parameter class	Dynamic: ALTER SESSION
Range of values	Any characters that can occur as part of a file name on the customer platform

TRACEFILE_IDENTIFIER specifies a custom identifier that becomes part of the Oracle Trace file name. Such a custom identifier is used to identify a trace file simply from its name and without having to open it or view its contents.

TRANSACTION_AUDITING

Parameter type	Boolean
Default value	`true`
Parameter class	Dynamic: `ALTER SYSTEM ... DEFERRED`
Range of values	`true | false`

If `TRANSACTION_AUDITING` is `true`, Oracle generates a special redo record that contains the user logon name, username, the session ID, some operating system information, and client information. For each successive transaction, Oracle generates a record that contains only the session ID. These subsequent records link back to the first record, which also contains the session ID.

TRANSACTIONS

Parameter type	Integer
Default value	Derived: (1.1 * `SESSIONS`)
Parameter class	Static
Range of values	4 to 2^{32}
Real Application Clusters	Multiple instances can have different values.

`TRANSACTIONS` specifies the maximum number of concurrent transactions. Greater values increase the size of the SGA and can increase the number of rollback segments allocated. The default value is greater than `SESSIONS` (and, in turn, `PROCESSES`) to allow for recursive transactions.

Note: You can set this parameter using `ALTER SYSTEM` only if you have started up the database using a server parameter file (spfile), and you must specify `SCOPE = SPFILE`.

TRANSACTIONS_PER_ROLLBACK_SEGMENT

Parameter type	Integer
Default value	`5`
Parameter class	Static

Range of values	1 to operating system-dependent
Real Application Clusters	Multiple instances can have different values.

TRANSACTIONS_PER_ROLLBACK_SEGMENT specifies the number of concurrent transactions you expect each rollback segment to have to handle. The minimum number of rollback segments acquired at startup is TRANSACTIONS divided by the value for this parameter. For example, if TRANSACTIONS is 101 and this parameter is 10, then the minimum number of rollback segments acquired would be the ratio 101 / 10, rounded up to 11.

UNDO_MANAGEMENT

Parameter type	String
Syntax	UNDO_MANAGEMENT = {MANUAL | AUTO}
Default value	MANUAL
Parameter class	Static
Real Application Clusters	Multiple instances must have the same value.

UNDO_MANAGEMENT specifies which undo space management mode the system should use. When set to AUTO, the instance starts in automatic undo management mode. In manual undo management mode, undo space is allocated externally as rollback segments.

UNDO_RETENTION

Parameter type	Integer
Default value	900
Parameter class	Dynamic: ALTER SYSTEM
Range of values	0 to 2^{32}-1 (max value represented by 32 bits)
Real Application Clusters	Multiple instances must have the same value.

UNDO_RETENTION specifies (in seconds) the amount of committed undo information to retain in the database. You can use UNDO_RETENTION to satisfy

queries that require old undo information to rollback changes to produce older images of data blocks. You can set the value at instance startup.

UNDO_SUPPRESS_ERRORS

Parameter type	Boolean
Default value	`false`
Parameter class	Dynamic: `ALTER SESSION`, `ALTER SYSTEM`
Range of values	`true | false`

`UNDO_SUPPRESS_ERRORS` enables users to suppress errors while executing manual undo management mode operations (for example, `ALTER ROLLBACK SEGMENT ONLINE`) in automatic undo management mode. Setting this parameter enables users to use the undo tablespace feature before all application programs and scripts are converted to automatic undo management mode. For example, if you have a tool that uses `SET TRANSACTION USE ROLLBACK SEGMENT` statement, you can add the statement "`ALTER SESSION SET UNDO_SUPPRESS_ERRORS = true`" to the tool to suppress the ORA-30019 error.

UNDO_TABLESPACE

Parameter type	String
Syntax	`UNDO_TABLESPACE =` *undoname*
Default value	The first available undo tablespace in the database.
Parameter class	Dynamic: `ALTER SYSTEM`
Range of values	Legal name of an existing undo tablespace
Real Application Clusters	Multiple instances can have different values.

`UNDO_TABLESPACE` specifies the undo tablespace to be used when an instance starts up. If this parameter is specified when the instance is in manual undo management mode, an error will occur and startup will fail.

USE_INDIRECT_DATA_BUFFERS

Parameter type	Boolean
Default value	`false`

Parameter class	Static
Range of values	`true | false`

USE_INDIRECT_DATA_BUFFERS controls how the system global area (SGA) uses memory. It enables or disables the use of the extended buffer cache mechanism for 32-bit platforms that can support more than 4 GB of physical memory. On platforms that do not support this much physical memory, this parameter is ignored.

USE_STORED_OUTLINES

Syntax: USE_STORED_OUTLINES = { TRUE | FALSE | *category_name*}

The USE_STORED_OUTLINES parameter determines whether the optimizer will use stored public outlines to generate execution plans. USE_STORED_OUTLINES is not an initialization parameter.

- TRUE causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- FALSE specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

USER_DUMP_DEST

Parameter type	String
Syntax	USER_DUMP_DEST = {*pathname* | *directory*}
Default value	Operating system-dependent
Parameter class	Dynamic: ALTER SYSTEM
Range of values	Any valid local path, directory, or disk

USER_DUMP_DEST specifies the pathname for a directory where the server will write debugging trace files on behalf of a user process.

UTL_FILE_DIR

Parameter type	String
Syntax	UTL_FILE_DIR = *pathname*

Default value	There is no default value.
Parameter class	Static
Range of values	Any valid directory path

UTL_FILE_DIR lets you specify one or more directories that Oracle should use for PL/SQL file I/O. If you are specifying multiple directories, you must repeat the UTL_FILE_DIR parameter for each directory on separate lines of the initialization parameter file.

WORKAREA_SIZE_POLICY

Parameter type	String
Syntax	WORKAREA_SIZE_POLICY = {AUTO | MANUAL}
Default value	If PGA_AGGREGATE_TARGET is set, then AUTO
	If PGA_AGGREGATE_TARGET is not set, then MANUAL
Parameter class	Dynamic: ALTER SESSION, ALTER SYSTEM

WORKAREA_SIZE_POLICY specifies the policy for sizing work areas. This parameter controls the mode in which working areas are tuned.

Examples

Archiving Redo Logs Manually: Examples The following statement manually archives the redo log file group with the log sequence number 4 in thread number 3:

```
ALTER SYSTEM ARCHIVE LOG THREAD 3 SEQUENCE 4;
```

The following statement manually archives the redo log file group containing the redo log entry with the SCN 9356083:

```
ALTER SYSTEM ARCHIVE LOG CHANGE 9356083;
```

The following statement manually archives the redo log file group containing a member named 'diskl:log6.log' to an archived redo log file in the location 'diska:[arch$]':

```
ALTER SYSTEM ARCHIVE LOG
    LOGFILE 'diskl:log6.log'
    TO 'diska:[arch$]';
```

Enabling Query Rewrite: Example This statement enables query rewrite in all sessions for all materialized views that have not been explicitly disabled:

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE;
```

Restricting Session Logons: Example You may want to restrict logons if you are performing application maintenance and you want only application developers with RESTRICTED SESSION system privilege to log on. To restrict logons, issue the following statement:

```
ALTER SYSTEM
   ENABLE RESTRICTED SESSION;
```

You can then terminate any existing sessions using the KILL SESSION clause of the ALTER SYSTEM statement.

After performing maintenance on your application, issue the following statement to allow any user with CREATE SESSION system privilege to log on:

```
ALTER SYSTEM
   DISABLE RESTRICTED SESSION;
```

Clearing the Shared Pool: Example You might want to clear the shared pool before beginning performance analysis. To clear the shared pool, issue the following statement:

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

Forcing a Checkpoint: Example The following statement forces a checkpoint:

```
ALTER SYSTEM CHECKPOINT;
```

Enabling Resource Limits: Example This ALTER SYSTEM statement dynamically enables resource limits:

```
ALTER SYSTEM SET RESOURCE_LIMIT = TRUE;
```

Changing Shared Server Settings: Examples The following statement changes the minimum number of shared server processes to 25:

```
ALTER SYSTEM SET SHARED_SERVERS = 25;
```

If there are currently fewer than 25 shared server processes, then Oracle creates more. If there are currently more than 25, then Oracle terminates some of them when they are finished processing their current calls if the load could be managed by the remaining 25.

The following statement dynamically changes the number of dispatcher processes for the TCP/IP protocol to 5 and the number of dispatcher processes for the DECNet protocol to 10:

```
ALTER SYSTEM
   SET DISPATCHERS =
      '(INDEX=0)(PROTOCOL=TCP)(DISPATCHERS=5)',
      '(INDEX=1)(PROTOCOL=DECNet)(DISPATCHERS=10)';
```

If there are currently fewer than 5 dispatcher processes for TCP, then Oracle creates new ones. If there are currently more than 5, then Oracle terminates some of them after the connected users disconnect.

If there are currently fewer than 10 dispatcher processes for DECNet, then Oracle creates new ones. If there are currently more than 10, then Oracle terminates some of them after the connected users disconnect.

If there are currently existing dispatchers for another protocol, then the preceding statement does not affect the number of dispatchers for that protocol.

Changing Licensing Parameters: Examples The following statement dynamically changes the limit on sessions for your instance to 64 and the warning threshold for sessions on your instance to 54:

```
ALTER SYSTEM
   SET LICENSE_MAX_SESSIONS = 64
   LICENSE_SESSIONS_WARNING = 54;
```

If the number of sessions reaches 54, then Oracle writes a warning message to the ALERT file for each subsequent session. Also, users with RESTRICTED SESSION system privilege receive warning messages when they begin subsequent sessions.

If the number of sessions reaches 64, then only users with RESTRICTED SESSION system privilege can begin new sessions until the number of sessions falls below 64 again.

The following statement dynamically disables the limit for sessions on your instance. After you issue the preceding statement, Oracle no longer limits the number of sessions on your instance.

```
ALTER SYSTEM SET LICENSE_MAX_SESSIONS = 0;
```

The following statement dynamically changes the limit on the number of users in the database to 200. After you issue the preceding statement, Oracle prevents the number of users in the database from exceeding 200.

```
ALTER SYSTEM SET LICENSE_MAX_USERS = 200;
```

Forcing a Log Switch: Example You may want to force a log switch to drop or rename the current redo log file group or one of its members, because you cannot drop or rename a file while Oracle is writing to it. The forced log switch affects only your instance's redo log thread. The following statement forces a log switch:

```
ALTER SYSTEM SWITCH LOGFILE;
```

Enabling Distributed Recovery: Example The following statement enables distributed recovery:

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

You may want to disable distributed recovery for demonstration or testing purposes.You can disable distributed recovery in both single-process and multiprocess mode with the following statement:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

When your demonstration or testing is complete, you can then enable distributed recovery again by issuing an ALTER SYSTEM statement with the ENABLE DISTRIBUTED RECOVERY clause.

Killing a Session: Example You may want to kill the session of a user that is holding resources needed by other users. The user receives an error message indicating that the session has been killed. That user can no longer make calls to the database without beginning a new session. Consider this data from the V$SESSION dynamic performance table:

```
SELECT sid, serial#, username
   FROM v$session;

  SID   SERIAL# USERNAME
----- --------- ----------------
    1         1
    2         1
    3         1
    4         1
    5         1
    7         1
    8        28 OPS$BQUIGLEY
   10       211 OPS$SWIFT
   11        39 OPS$OBRIEN
   12        13 SYSTEM
```

```
   13         8 SCOTT
```

The following statement kills the session of the user `scott` using the `SID` and `SERIAL#` values from `V$SESSION`:

```
ALTER SYSTEM KILL SESSION '13, 8';
```

Disconnecting a Session: Example The following statement disconnects user `scott`'s session, using the `SID` and `SERIAL#` values from `V$SESSION`:

```
ALTER SYSTEM DISCONNECT SESSION '13, 8' POST_TRANSACTION;
```

11

SQL Statements: ALTER TABLE to ALTER TABLESPACE

This chapter contains the following SQL statements:

- ALTER TABLE
- ALTER TABLESPACE

ALTER TABLE

Purpose

Use the ALTER TABLE statement to alter the definition of a nonpartitioned table, a partitioned table, a table partition, or a table subpartition. For object tables or relational tables with object columns, use ALTER TABLE to convert the table to the latest definition of its referenced type after the type has been altered.

Prerequisites

The table must be in your own schema, or you must have ALTER privilege on the table, or you must have ALTER ANY TABLE system privilege. For some operations you may also need the CREATE ANY INDEX privilege.

Additional Prerequisites for Partitioning Operations If you are not the owner of the table, then you need the DROP ANY TABLE privilege in order to use the *drop_table_partition* or *truncate_table_partition* clause.

You must also have space quota in the tablespace in which space is to be acquired in order to use the *add_table_partition*, *modify_table_partition*, *move_table_partition*, and *split_table_partition* clauses.

Additional Prerequisites for Constraints and Triggers To enable a unique or primary key constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle creates an index on the columns of the unique or primary key in the schema containing the table.

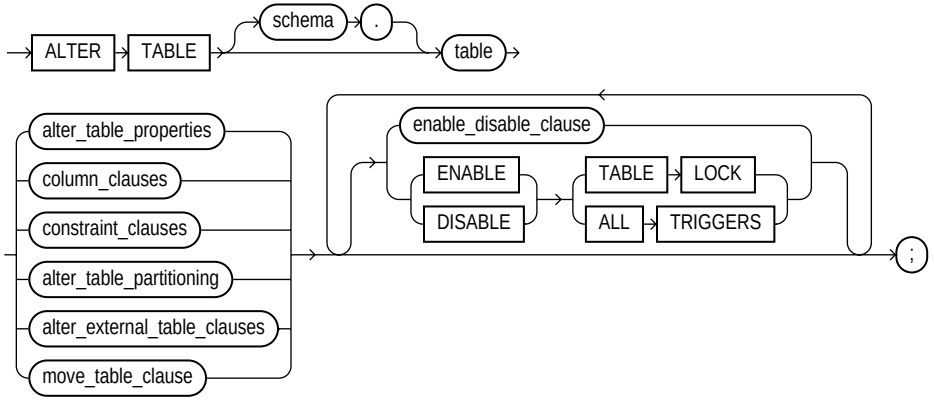
To enable or disable triggers, the triggers must be in your schema or you must have the ALTER ANY TRIGGER system privilege.

Additional Prerequisites When Using Object Types To use an object type in a column definition when modifying a table, either that object must belong to the same schema as the table being altered, or you must have either the EXECUTE ANY TYPE system privilege or the EXECUTE schema object privilege for the object type.

> **See Also:** CREATE INDEX on page 13-62 for information on the privileges needed to create indexes

Syntax

alter_table::=

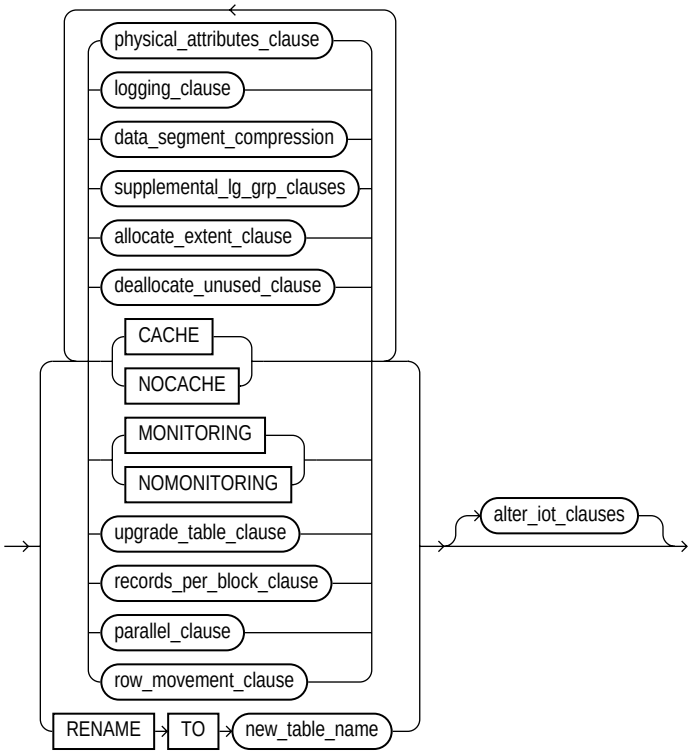


Groups of ALTER TABLE syntax:

- **alter_table_properties::=** on page 11-4
- **column_clauses::=** on page 11-9
- **constraint_clauses::=** on page 11-11
- **alter_table_partitioning::=** on page 11-17
- **alter_external_table_clause::=** on page 11-16
- **move_table_clause::=** on page 11-29
- **enable_disable_clause::=** on page 11-29

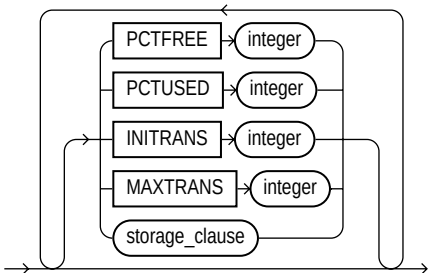
After each clause you will find links additional links to its component subclauses.

alter_table_properties::=



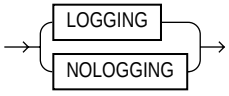
(*physical_attributes_clause::=* on page 11-5, *logging_clause::=* on page 7-46, *data_segment_compression::=* on page 11-5, *supplemental_lg_grp_clauses::=* on page 11-5, *allocate_extent_clause::=* on page 11-5,*deallocate_unused_clause::=* on page 11-6, *upgrade_table_clause::=* on page 11-6, *records_per_block_clause::=* on page 11-6, *parallel_clause::=* on page 11-6, *row_movement_clause::=* on page 11-6, *alter_iot_clauses::=* on page 11-7)

physical_attributes_clause::=

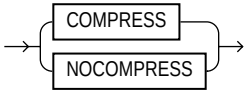


(*storage_clause* on page 7-56)

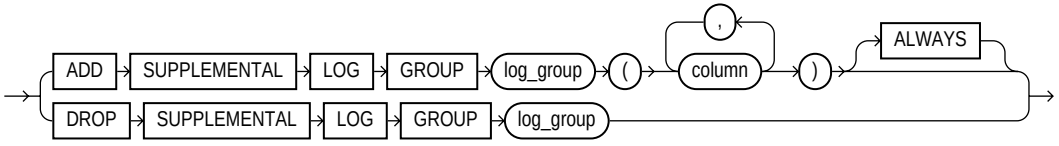
logging_clause::=



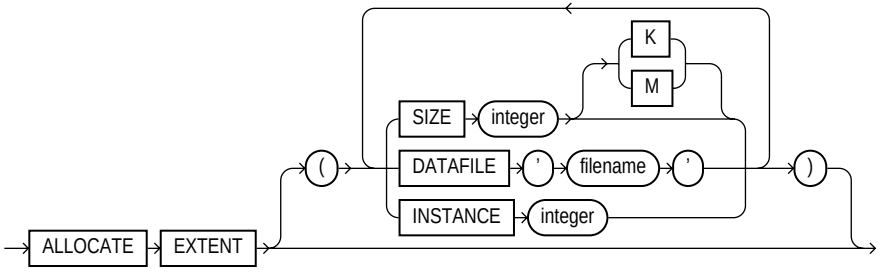
data_segment_compression::=



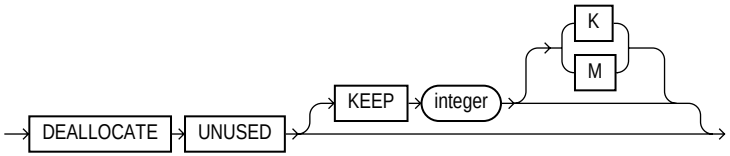
supplemental_lg_grp_clauses::=



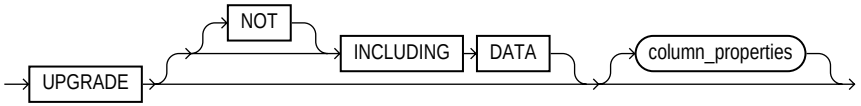
allocate_extent_clause::=



deallocate_unused_clause::=

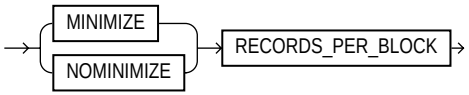


upgrade_table_clause::=

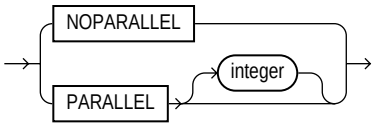


(*column_properties::=* on page 11-11, *modify_LOB_storage_clause::=* on page 11-14)

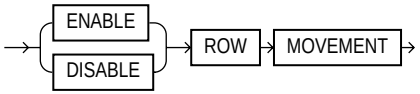
records_per_block_clause::=



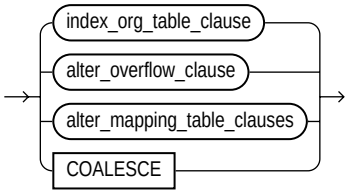
parallel_clause::=



row_movement_clause::=

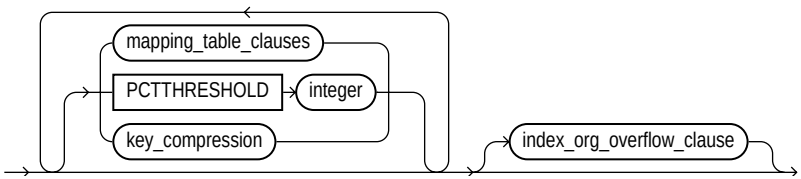


alter_iot_clauses::=

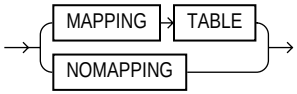


(*alter_overflow_clause::=* on page 11-8, *alter_mapping_table_clause::=* on page 11-8)

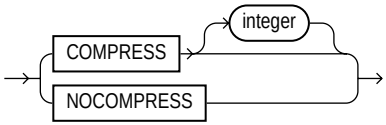
index_org_table_clause::=



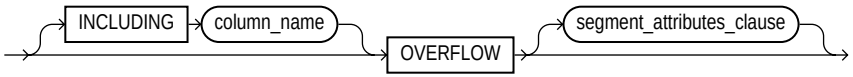
mapping_table_clauses::=



key_compression::=

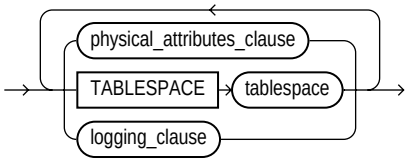


index_org_overflow_clause::=



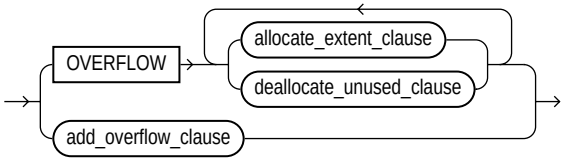
(*segment_attributes_clause::=* on page 11-8)

segment_attributes_clause::=



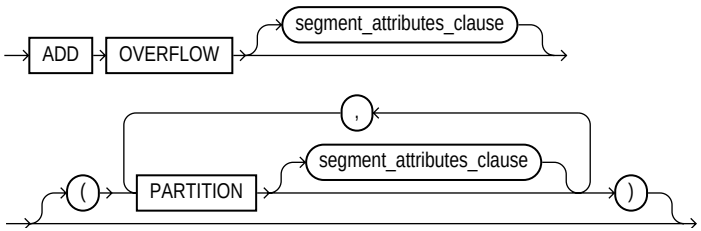
(*physical_attributes_clause::=* on page 11-5, *logging_clause* on page 7-45)

alter_overflow_clause::=



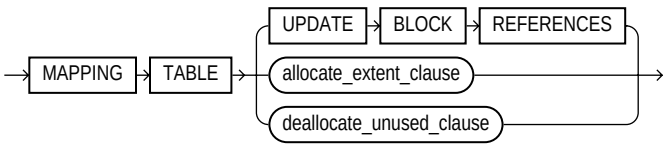
(*segment_attributes_clause::=* on page 11-8, *allocate_extent_clause::=* on page 11-5, *deallocate_unused_clause::=* on page 11-6)

add_overflow_clause::=



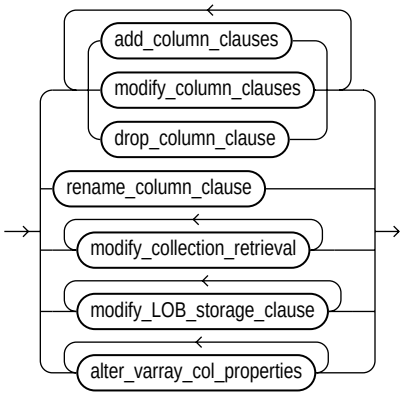
(*segment_attributes_clause::=* on page 11-8)

alter_mapping_table_clause::=

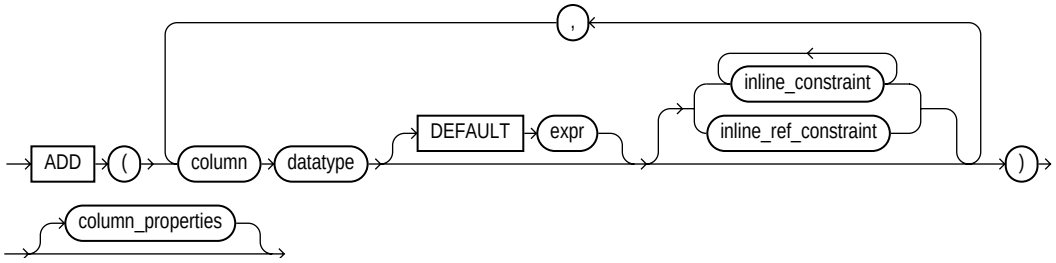


(*allocate_extent_clause::=* on page 11-5, *deallocate_unused_clause::=* on page 11-6)

column_clauses::=

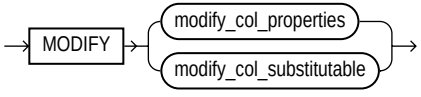


add_column_clause::=

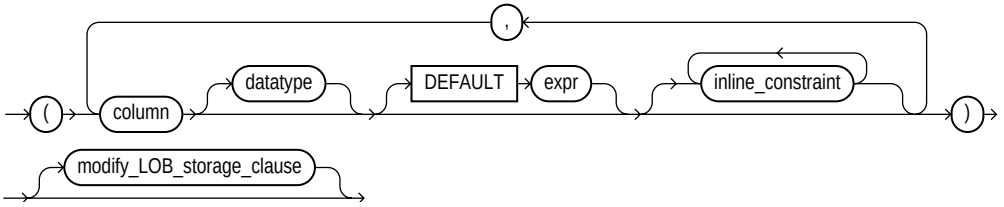


(*inline_constraint* and *inline_ref_constraint*: *constraints* on page 7-5, *column_properties::=* on page 11-11)

modify_column_clause::=

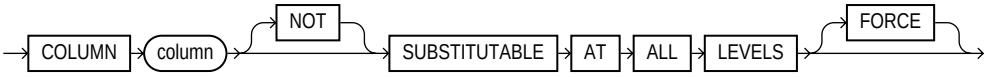


modify_col_properties::=

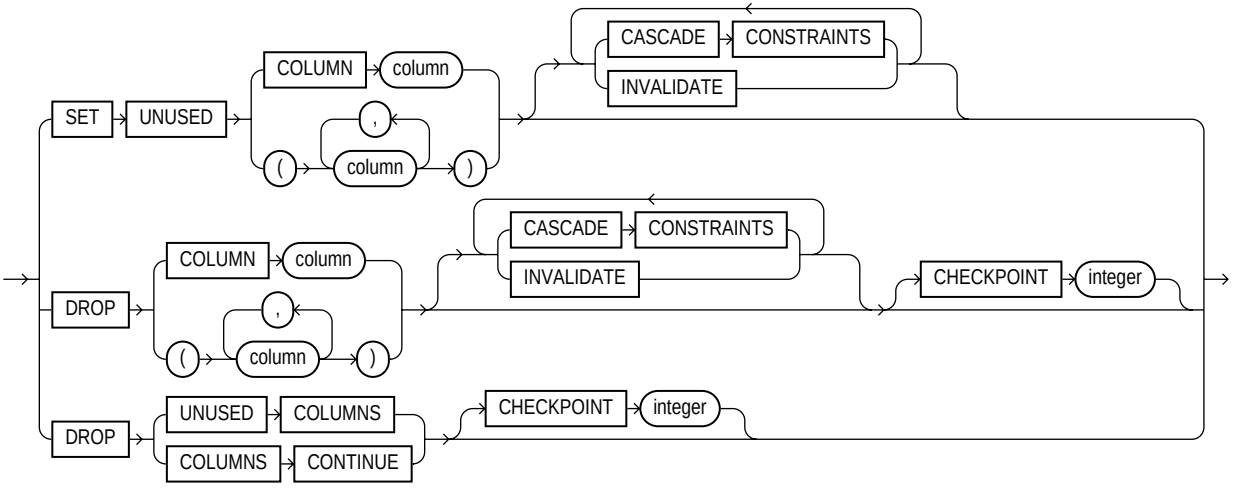


(*inline_constraint*: *constraints* on page 7-5, *modify_LOB_storage_clause::=* on page 11-14)

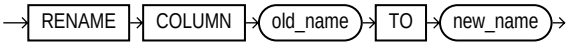
modify_col_substitutable::=



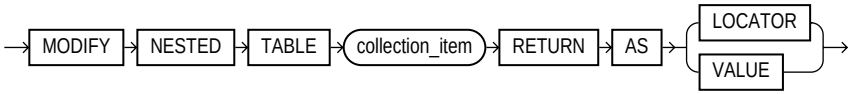
drop_column_clause::=



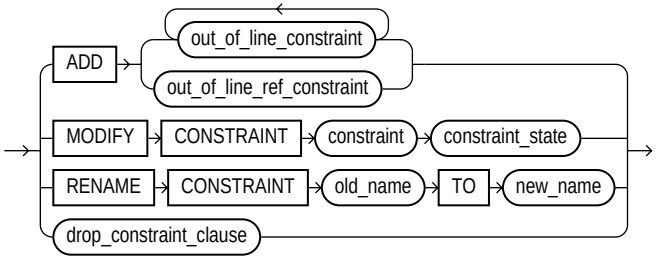
rename_column_clause::=



modify_collection_retrieval::=

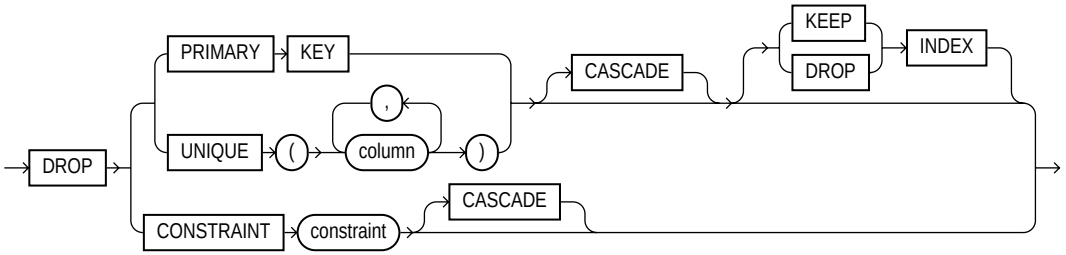


constraint_clauses::=

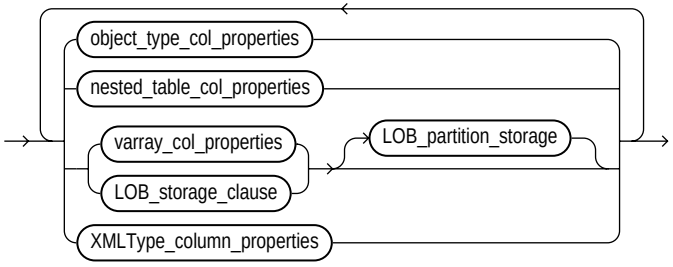


(constraint_state: *constraints* on page 7-5)

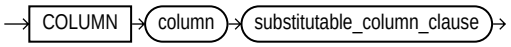
drop_constraint_clause::=



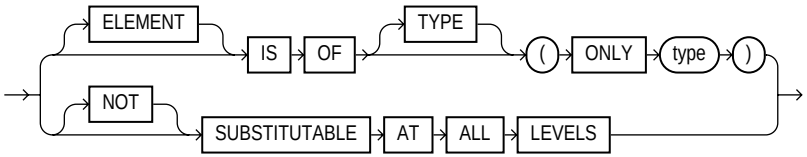
column_properties::=



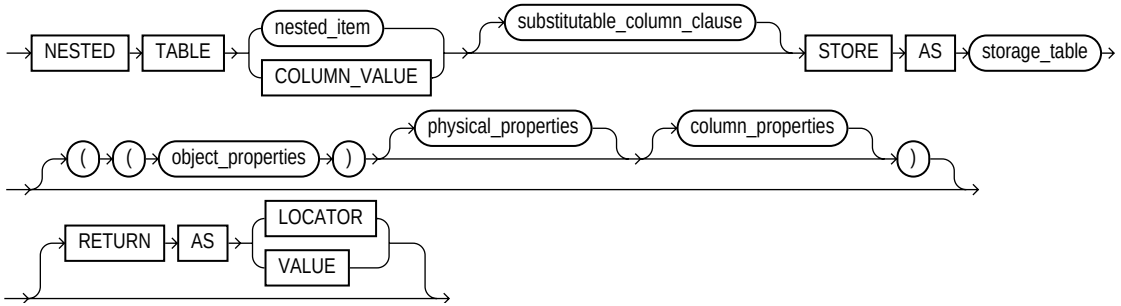
object_type_col_properties::=



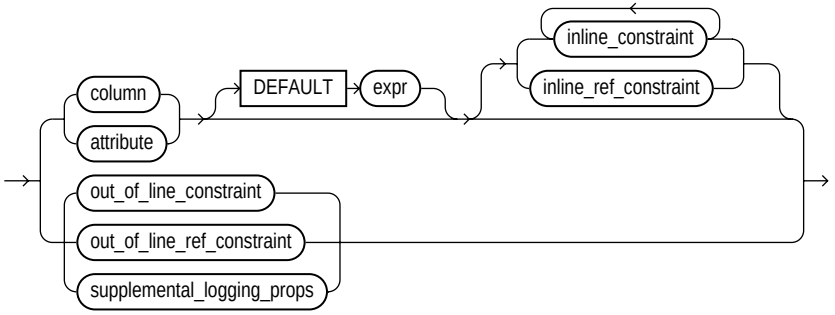
substitutable_column_clause::=



nested_table_col_properties::=

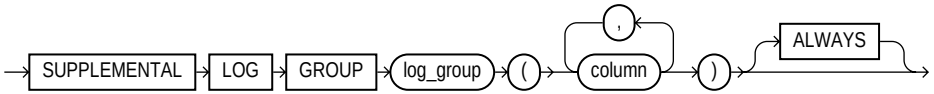


object_properties::=

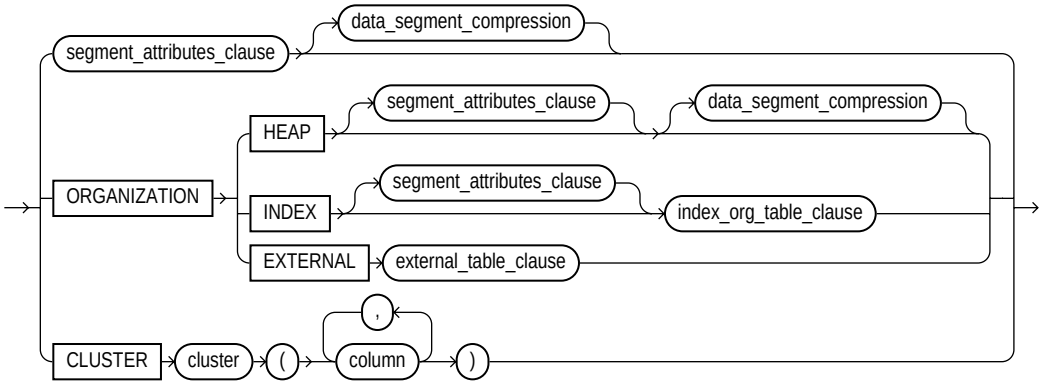


(*inline_constraint*, *inline_ref_constraint*, *out_of_line_constraint*, *out_of_line_ref_constraint*: *constraints* on page 7-5)

supplemental_logging_props::=

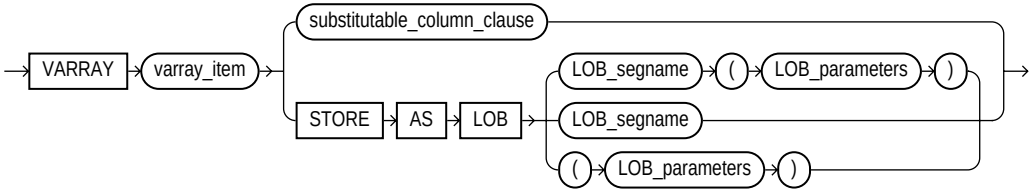


physical_properties::=



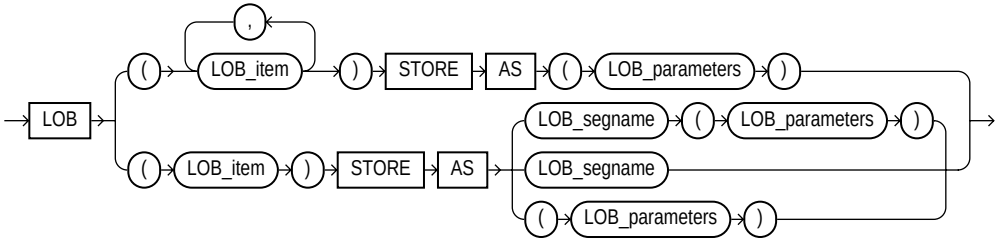
(*segment_attributes_clause::=* on page 11-8, *index_org_table_clause::=* on page 11-7, *external_data_properties::=* on page 11-17)

varray_col_properties::=

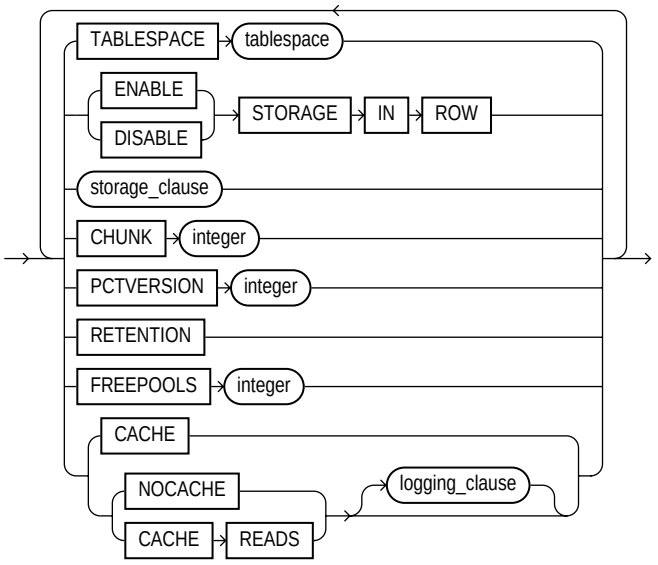


(*substitutable_column_clause::=* on page 11-12)

LOB_storage_clause::=

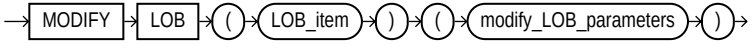


LOB_parameters::=

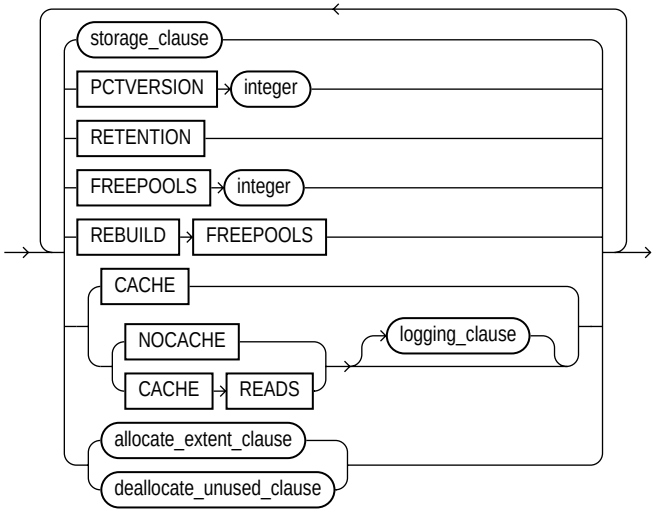


(*storage_clause::=* on page 7-58, *logging_clause::=* on page 7-46)

modify_LOB_storage_clause::=

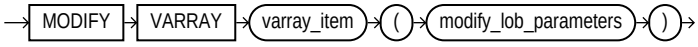


modify_LOB_parameters::=

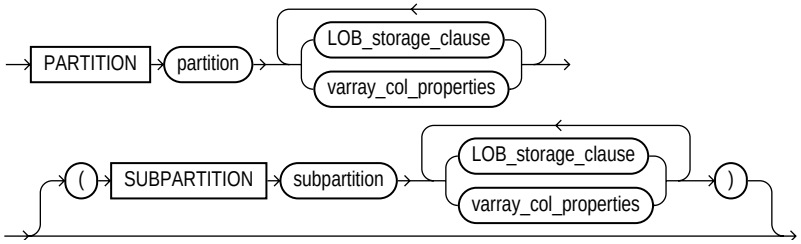


(*storage_clause::=* on page 7-58, *logging_clause::=* on page 7-46, *allocate_extent_clause::=* on page 11-5, *deallocate_unused_clause::=* on page 11-6)

alter_varray_col_properties::=

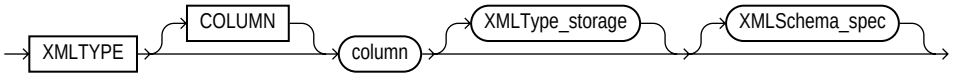


LOB_partition_storage::=

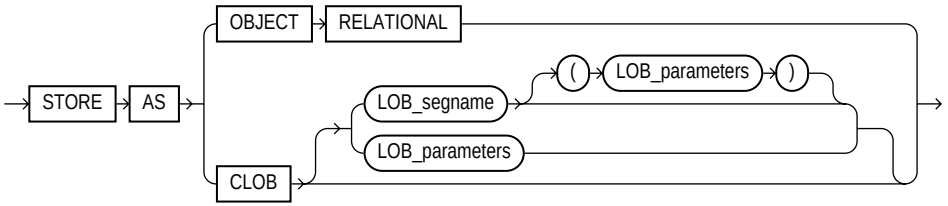


(*LOB_storage_clause::=* on page 11-14, *varray_col_properties::=* on page 11-13)

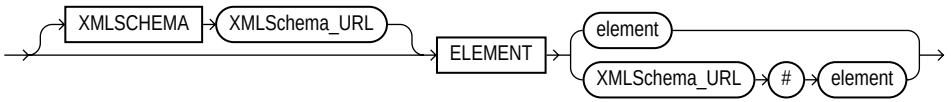
XMLType_column_properties::=



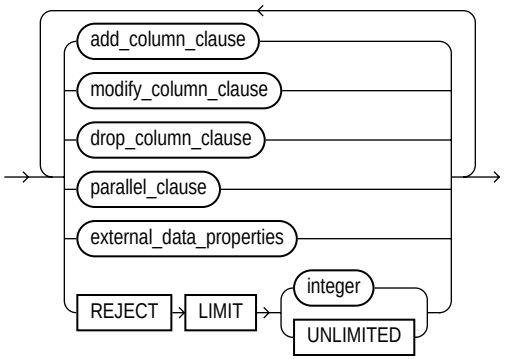
XMLType_storage::=



XMLSchema_spec::=

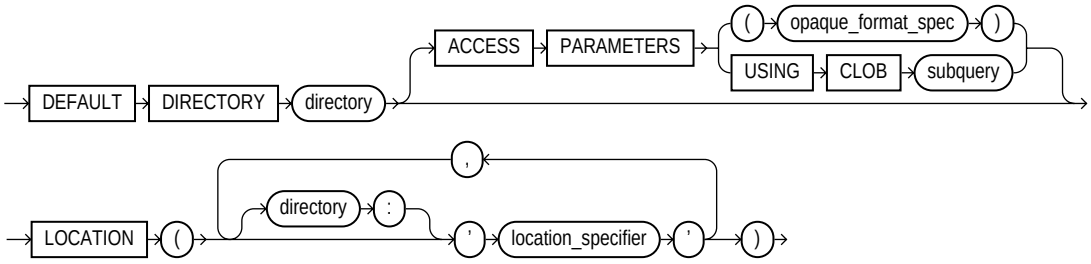


alter_external_table_clause::=

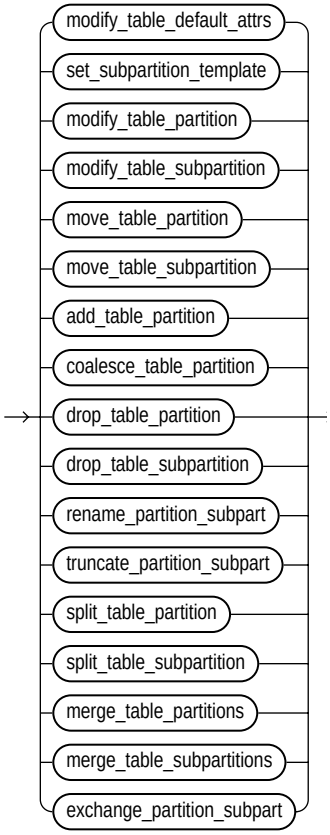


(*add_column_clause::=* on page 11-9, *modify_column_clause::=* on page 11-9, *drop_column_clause::=* on page 11-10, *drop_constraint_clause::=* on page 11-11, *parallel_clause::=* on page 11-28)

external_data_properties::=



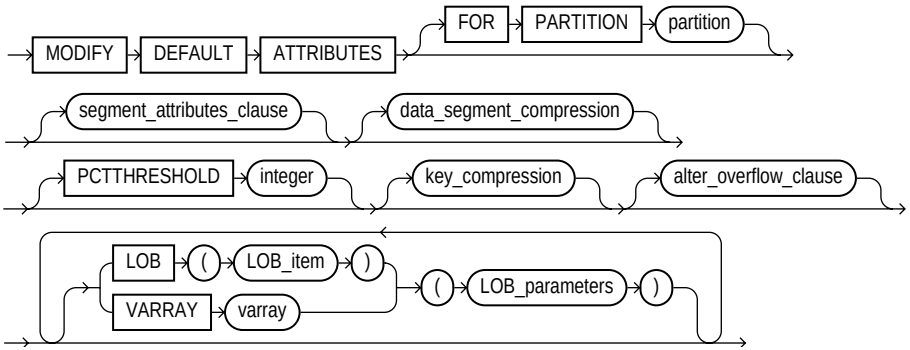
alter_table_partitioning::=



(*modify_table_default_attrs::=* on page 11-18, *set_subpartition_template::=* on page 11-18, *modify_table_partition::=* on page 11-19,

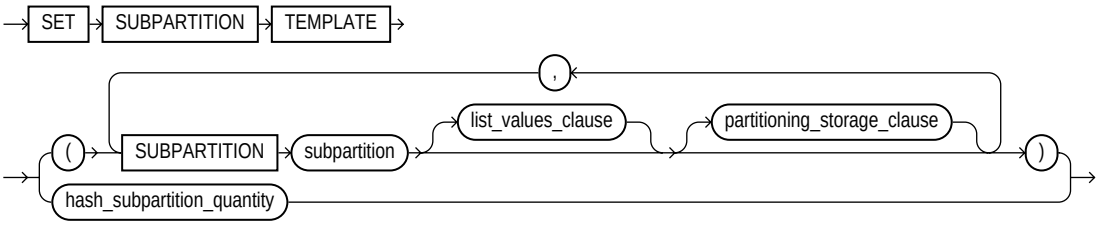
modify_table_subpartition::= on page 11-20, *move_table_partition::=* on page 11-20, *move_table_subpartition::=* on page 11-20, *add_table_partition::=* on page 11-21, *coalesce_table_partition::=* on page 11-21, *drop_table_partition::=* on page 11-22, *drop_table_subpartition::=* on page 11-22, *rename_partition_subpart::=* on page 11-22, *truncate_partition_subpart::=* on page 11-22, *split_table_partition::=* on page 11-23, *split_table_subpartition::* on page 11-23, *merge_table_partitions::=* on page 11-23, *merge_table_subpartitions::=* on page 11-24, *exchange_partition_subpart::=* on page 11-24

modify_table_default_attrs::=



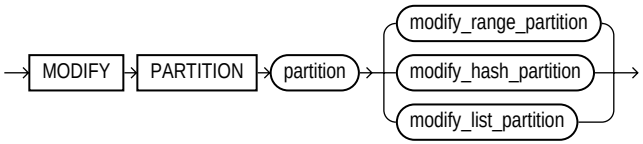
(*segment_attributes_clause::=* on page 11-8, *key_compression::= on page 11-7*, *LOB_parameters::=* on page 11-14, *alter_overflow_clause::=* on page 11-8)

set_subpartition_template::=



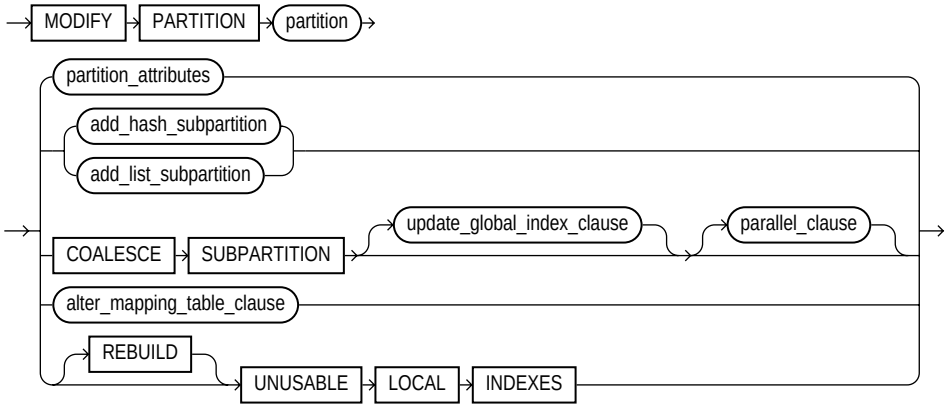
(*list_values_clause::=* on page 11-24, *partitioning_storage_clause::=* on page 11-25)

modify_table_partition::=



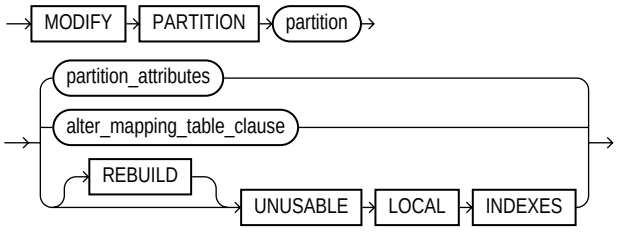
(*modify_range_partition::=* on page 11-19, *modify_hash_partition::=* on page 11-19, *modify_list_partition::=* on page 11-20)

modify_range_partition::=



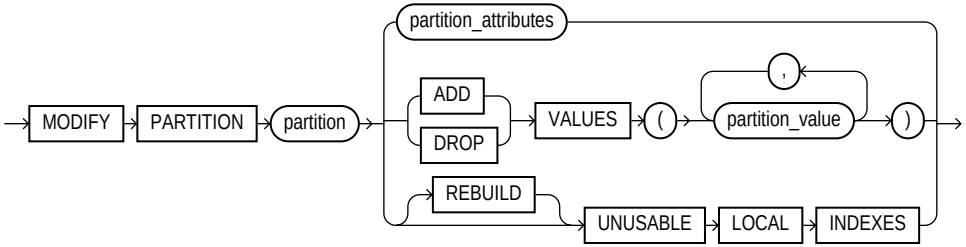
(*partition_attributes::=* on page 11-25, *alter_mapping_table_clause::=* on page 11-8)

modify_hash_partition::=



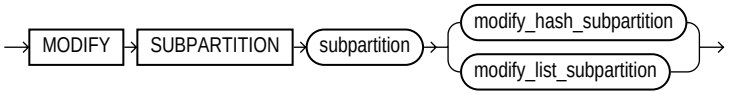
(*partition_attributes::=* on page 11-25, *add_hash_subpartition::=* on page 11-26, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28, *alter_mapping_table_clause::=* on page 11-8)

modify_list_partition::=



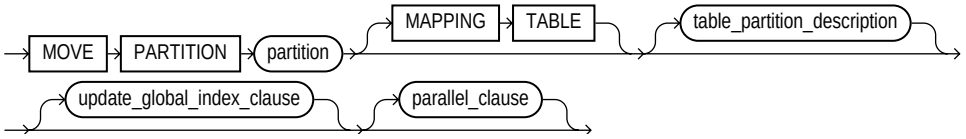
(*partition_attributes::=* on page 11-25, *add_list_subpartition::=* on page 11-26)

modify_table_subpartition::=



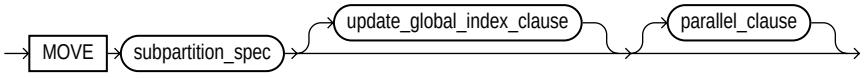
modify_hash_subpartition::= on page 11-26, *modify_list_subpartition::=* on page 11-27)

move_table_partition::=



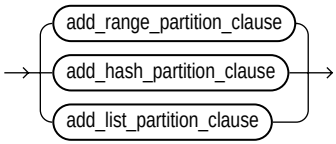
(*table_partition_description::=* on page 11-27, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

move_table_subpartition::=

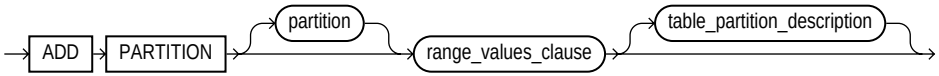


(*subpartition_spec::=* on page 11-28, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

add_table_partition::=

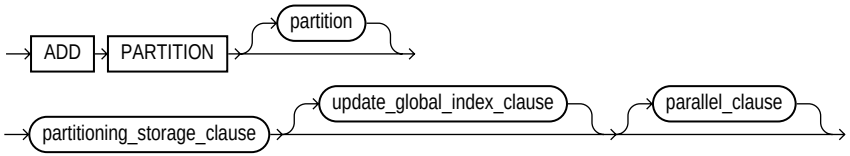


add_range_partition_clause::=



(*range_values_clause::=* on page 11-25, *table_partition_description::=* on page 11-27)

add_hash_partition_clause::=

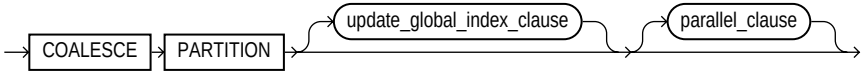


(*partitioning_storage_clause::=* on page 11-25, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

add_list_partition_clause::=

(*list_values_clause::=* on page 11-24, *table_partition_description::=* on page 11-27)

coalesce_table_partition::=



(*update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

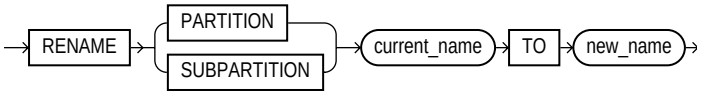
drop_table_partition::=

(*update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

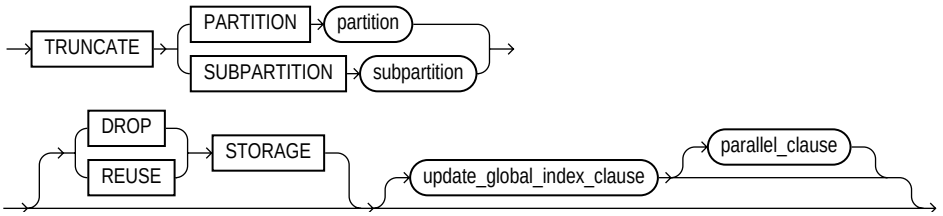
drop_table_subpartition::=

(*update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

rename_partition_subpart::=

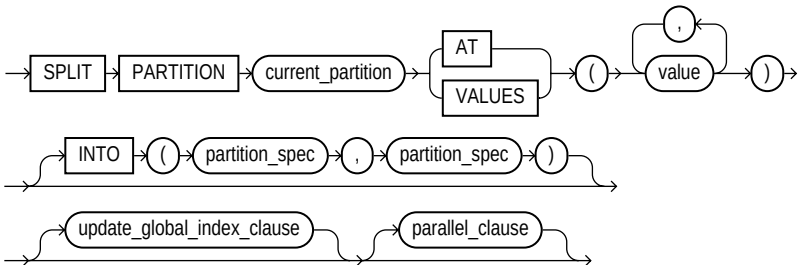


truncate_partition_subpart::=



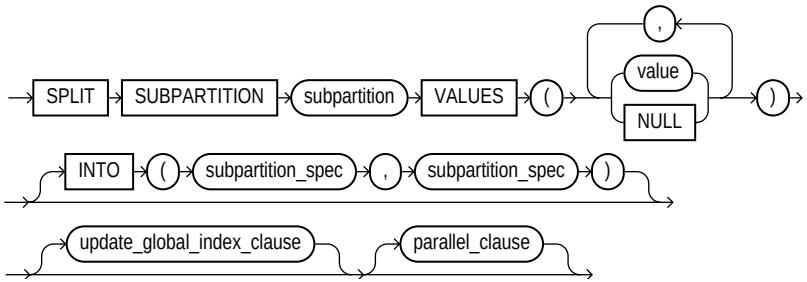
(*update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

split_table_partition::=



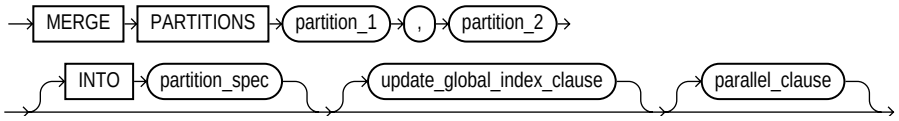
(*partition_spec::=* on page 11-28, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

split_table_subpartition::



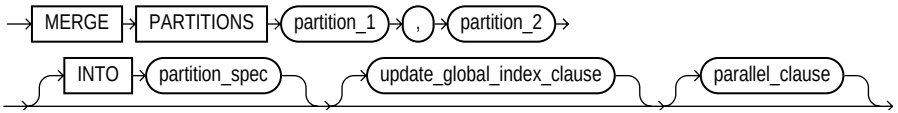
(*subpartition_spec::=* on page 11-28, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

merge_table_partitions::=



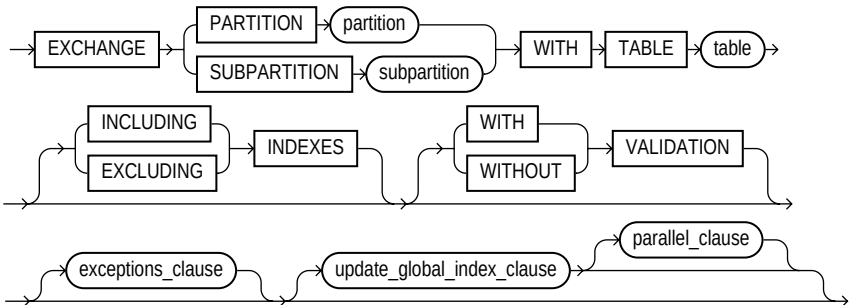
(*partition_spec::=* on page 11-28, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

merge_table_subpartitions::=



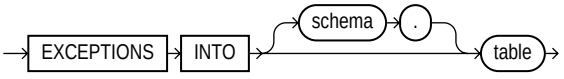
(*subpartition_spec::=* on page 11-28, *update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

exchange_partition_subpart::=

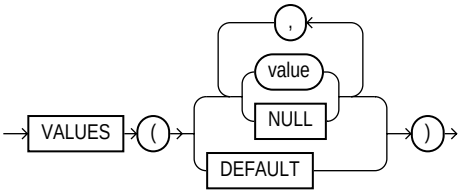


(*update_global_index_clause::=* on page 11-28, *parallel_clause::=* on page 11-28)

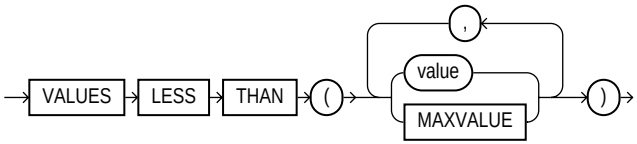
exceptions_clause::=



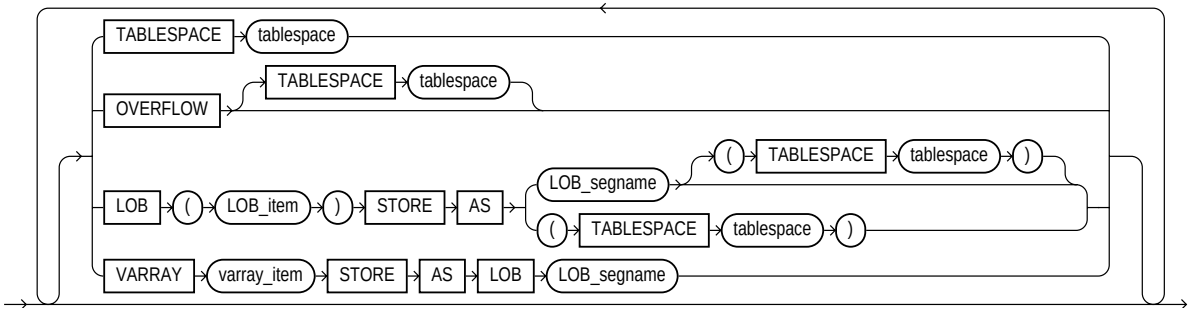
list_values_clause::=



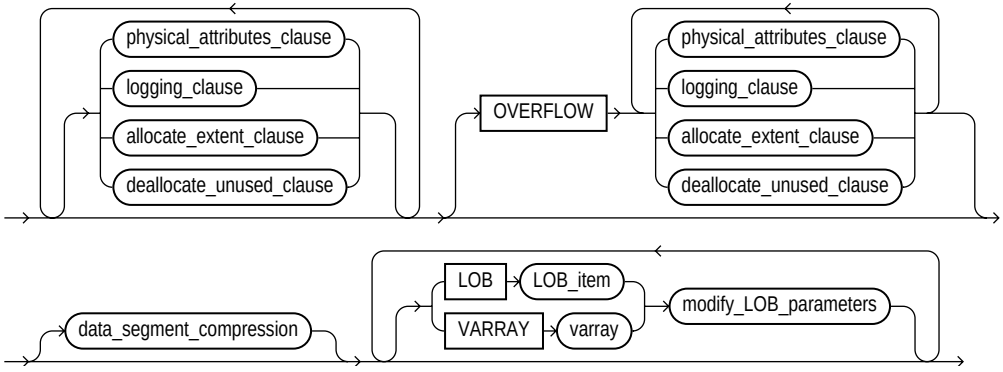
range_values_clause::=



partitioning_storage_clause::=

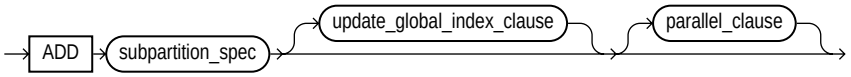


partition_attributes::=



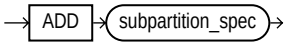
(*physical_attributes_clause::=* on page 11-5, *logging_clause::=* on page 7-46, *allocate_extent_clause::=* on page 11-5, *deallocate_unused_clause::=* on page 11-6, *data_segment_compression::=* on page 11-5, *modify_LOB_parameters::=* on page 11-15)

add_hash_subpartition::=



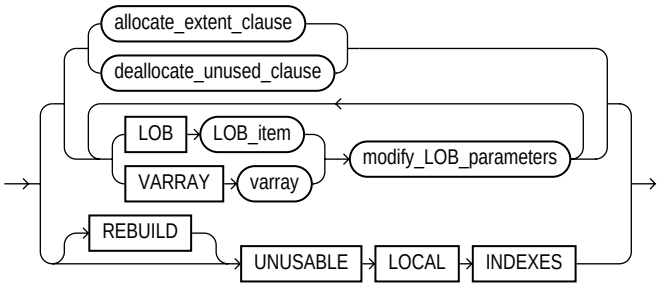
(*subpartition_spec::=* on page 11-28, *update_global_index_clause::=* on page 11-28)

add_list_subpartition::=



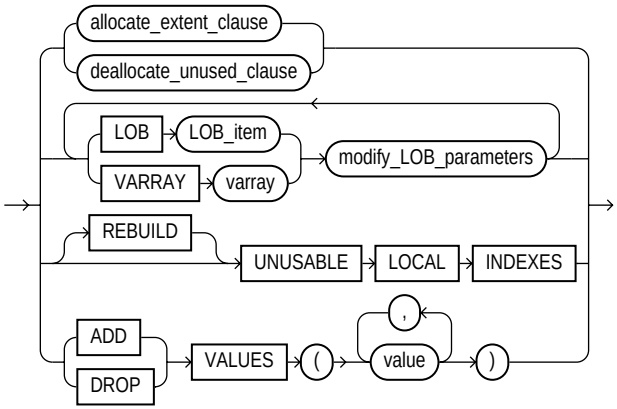
(*subpartition_spec::=* on page 11-28)

modify_hash_subpartition::=



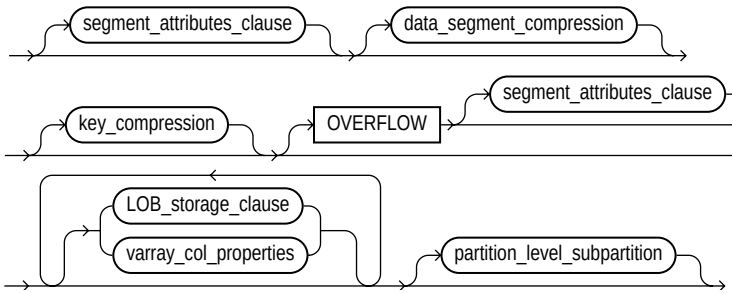
(*allocate_extent_clause::=* on page 11-5, *deallocate_unused_clause::=* on page 11-6, *modify_LOB_parameters::=* on page 11-15)

modify_list_subpartition::=



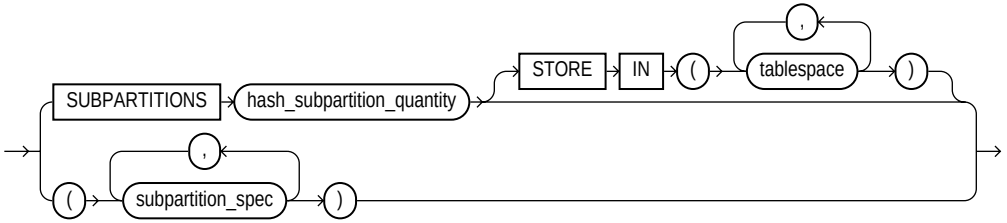
(*allocate_extent_clause::=* on page 11-5, *deallocate_unused_clause::=* on page 11-6, *modify_LOB_parameters::=* on page 11-15)

table_partition_description::=



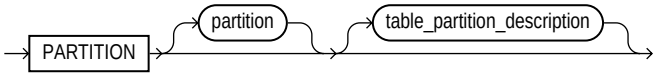
(*segment_attributes_clause::=* on page 11-8, *key_compression::=* on page 11-7, *LOB_storage_clause::=* on page 11-14, *varray_col_properties::=* on page 11-13)

partition_level_subpartition::=



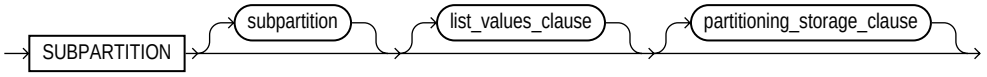
(*subpartition_spec::=* on page 11-28)

partition_spec::=



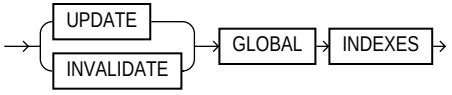
(*table_partition_description::=* on page 11-27)

subpartition_spec::=

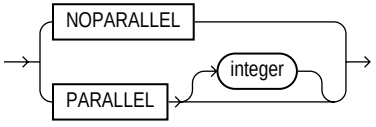


(*list_values_clause::=* on page 11-24, *partitioning_storage_clause::=* on page 11-25)

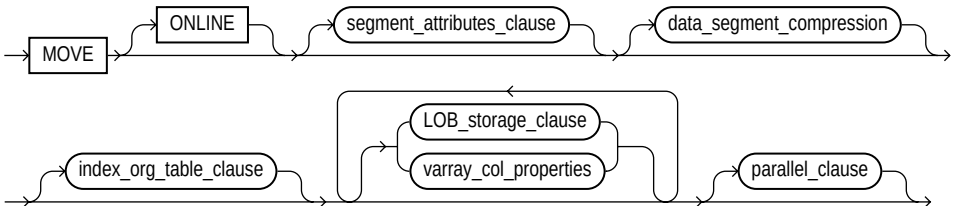
update_global_index_clause::=



parallel_clause::=

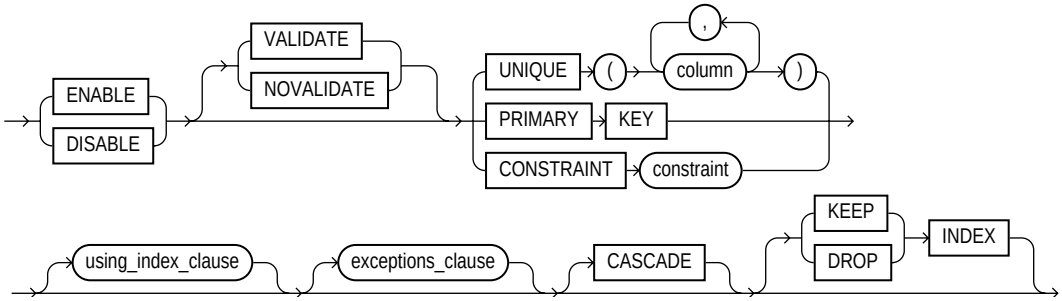


move_table_clause::=



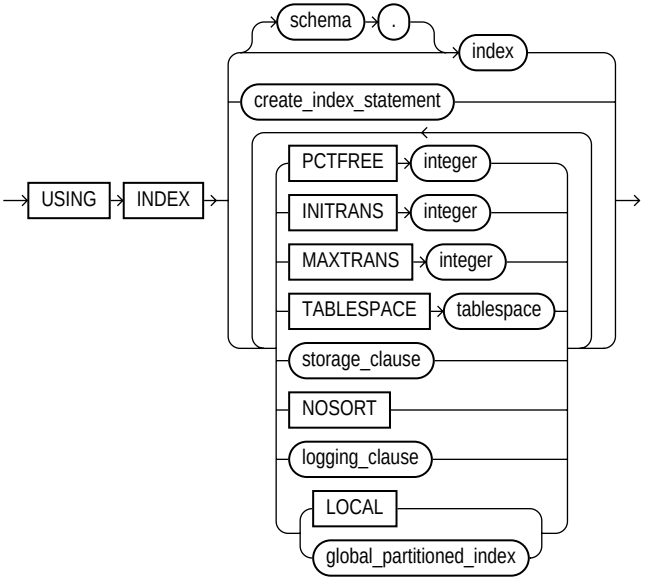
(*segment_attributes_clause::=* on page 11-8, *index_org_table_clause::=* on page 11-7, *LOB_storage_clause::=* on page 11-14, *varray_col_properties::=* on page 11-13)

enable_disable_clause::=



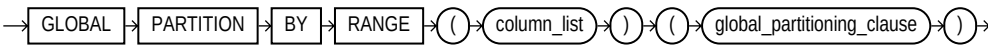
(*using_index_clause::=* on page 11-30, *exceptions_clause::=* on page 11-24,)

using_index_clause::=

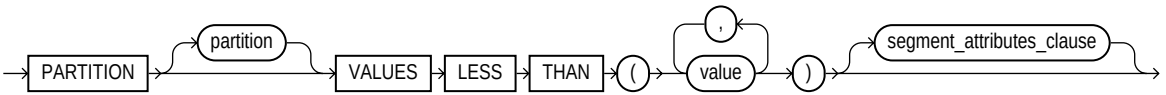


(*create_index::=* on page 13-63, *storage_clause* on page 7-56, *logging_clause::=* on page 7-46)

global_partitioned_index::=



index_partitioning_clause::=



Keywords and Parameters

Many clauses of the ALTER TABLE statement have the same functionality they have in a CREATE TABLE statement. For more information on such clauses, please see CREATE TABLE on page 15-7.

Note: Operations performed by the ALTER TABLE statement can cause Oracle to invalidate procedures and stored functions that access the table. For information on how and when Oracle invalidates such objects, see *Oracle9i Database Concepts*.

schema

Specify the schema containing the table. If you omit *schema*, then Oracle assumes the table is in your own schema.

table

Specify the name of the table to be altered.

Restrictions on Temporary Tables

You can modify, drop columns from, or rename a temporary table. However, for a temporary table you cannot:

- Add columns of nested table or varray type. You can add columns of other types.
- Specify referential integrity (foreign key) constraints for an added or modified column.
- Specify the following clauses of the *LOB_storage_clause* for an added or modified LOB column: TABLESPACE, *storage_clause*, *logging_clause*, or the *LOB_index_clause*.
- Specify the *physical_attributes_clause*, *nested_table_col_properties*, *parallel_clause*, *allocate_extent_clause*, *deallocate_unused_clause*, or any of the index organized table clauses.
- Exchange partitions between a partition and a temporary table.
- Specify the *logging_clause*.
- Specify MOVE.

Restrictions on External Tables:

You can add, drop, or modify the columns of an external table. However, for an external table you cannot:

- Add a LONG, LOB, or object type column or change the datatype of an external table column to any of these datatypes.

- Add a constraint to an external table.
- Modify the storage parameters of an external table.
- Specify the *logging_clause*.
- Specify MOVE.

Note: If you alter a table that is a master table for one or more materialized views, then Oracle marks the materialized views INVALID. Invalid materialized views cannot be used by query rewrite and cannot be refreshed. For information on revalidating a materialized view, see ALTER MATERIALIZED VIEW on page 9-92.

See Also: *Oracle9i Data Warehousing Guide* for more information on materialized views in general

alter_table_properties

Use the *alter_table_clauses* to modify a database table.

physical_attributes_clause

The *physical_attributes_clause* lets you change the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and storage characteristics.

Restrictions on the *physical_attributes_clause***:**

- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
- If you attempt to alter the storage attributes of tables in locally managed tablespaces, then Oracle raises an error. However, if some segments of a partitioned table reside in a locally managed tablespace and other segments reside in a dictionary-managed tablespace, then Oracle alters the storage attributes of the segments in the dictionary-managed tablespace but does not alter the attributes of the segments in the locally managed tablespace, and does not raise an error.
- For segments with automatic segment-space management, Oracle ignores attempts to change the PCTUSED setting. If you alter the PCTFREE setting, then you must subsequently run the DBMS_REPAIR.segment_fix_status

procedure to implement the new setting on blocks already allocated to the segment.

Cautions:

- For a nonpartitioned table, the values you specify override any values specified for the table at create time.
- For a range-, list-, or hash-partitioned table, the values you specify are the default values for the table and the actual values for every existing partition, overriding any values already set for the partitions. To change default table attributes without overriding existing partition values, use the *modify_table_default_attrs* clause.
- For a composite-partitioned table, the values you specify are the default values for the table and all partitions of the table and the actual values for all subpartitions of the table, overriding any values already set for the subpartitions. To change default partition attributes without overriding existing subpartition values, use the *modify_table_default_attrs* clause with the FOR PARTITION clause.

See Also:

- *physical_attributes_clause* on page 7-52 for a full description of the physical attribute parameters
- *storage_clause* on page 7-56 for a description of storage parameters

data_segment_compression

Use the *data_segment_compression* clause to instruct Oracle whether to compress data segments to reduce disk and memory use. The COMPRESS keyword enables data segment compression. The NOCOMPRESS keyword disables data segment compression.

Note: The first time a table is altered in such a way that compressed data will be added, all bitmap indexes and bitmap index partitions on that table must be marked UNUSABLE.

See Also:

- *Oracle9i Database Performance Tuning Guide and Reference* for information on calculating the compression ratio and to *Oracle9i Data Warehousing Guide* for information on data compression usage scenarios
- *data_segment_compression* clause of CREATE TABLE on page 15-29 information on creating objects with data segment compression

logging_clause

Specify whether subsequent Direct Loader (SQL*Loader) and direct-path INSERT operations against a nonpartitioned table, table partition, all partitions of a partitioned table, or all subpartitions of a partition will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

When used with the *modify_table_default_attrs* clause, this clause affects the logging attribute of a partitioned table.

The *logging_clause* also specifies whether ALTER TABLE ... MOVE and ALTER TABLE ... SPLIT operations will be logged or not logged.

See Also:

- *logging_clause* on page 7-45 for a full description of this clause
- *Oracle9i Data Warehousing Guide* for more information about the *logging_clause* and parallel DML

supplemental_lg_grp_clauses

The *supplemental_lg_grp_clauses* let you add and drop supplemental redo log groups.

- Use the ADD LOG GROUP clause to add a redo log group.
- Use the DROP LOG GROUP clause to drop a redo log group when it is no longer needed.

See Also: *Oracle9i Data Guard Concepts and Administration* for information on supplemental redo log groups

allocate_extent_clause

Use the *allocate_extent_clause* to explicitly allocate a new extent for the table, the partition or subpartition, the overflow data segment, the LOB data segment, or the LOB index.

Restriction on the *allocate_extent_clause***:** You cannot allocate an extent for a temporary table or for a range- or composite-partitioned table.

> **See Also:** *allocate_extent_clause* on page 7-2 for a full description of this clause and "Allocating Extents: Example" on page 11-96

deallocate_unused_clause

Use the *deallocate_unused_clause* to explicitly deallocate unused space at the end of the table, partition or subpartition, overflow data segment, LOB data segment, or LOB index and make the space available for other segments in the tablespace.

> **See Also:** *deallocate_unused_clause* on page 7-37 for a full description of this clause and "Deallocating Unused Space: Example" on page 11-91

CACHE | NOCACHE

CACHE Clause For data that is accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the *most recently used* end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

As a parameter in the *LOB_storage_clause*, CACHE specifies that Oracle places LOB data values in the buffer cache for faster access.

Restriction: You cannot specify CACHE for an index-organized table. However, index-organized tables implicitly provide CACHE behavior.

NOCACHE Clause For data that is not accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the *least recently used* end of the LRU list in the buffer cache when a full table scan is performed.

As a parameter in the *LOB_storage_clause*, NOCACHE specifies that the LOB value is either not brought into the buffer cache or brought into the buffer cache and placed at the least recently used end of the LRU list. (The latter is the default behavior.) NOCACHE is the default for LOB storage.

Restriction: You cannot specify NOCACHE for index-organized tables.

MONITORING | NOMONITORING

MONITORING Clause Specify MONITORING if you want Oracle to collect modification statistics on *table*. These statistics are estimates of the number of rows affected by DML statements over a particular period of time. They are available for use by the optimizer or for analysis by the user.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for more information on using this clause

NOMONITORING Clause Specify NOMONITORING if you do not want Oracle to collect modification statistics on *table*.

Restriction on MONITORING: You cannot specify MONITORING or NOMONITORING for a temporary table.

upgrade_table_clause

The *upgrade_table_clause* is relevant for object tables and for relational tables with object columns. It lets you instruct Oracle to convert the metadata of the target table to conform with the latest version of each referenced type. If table is already valid, then the table metadata remains unchanged.

Restriction on the *upgrade_table_clause***:** Within this clause, you cannot specify *object_type_col_properties* as a clause of *column_properties*.

INCLUDING DATA Specify INCLUDING DATA if you want Oracle to convert the data in the table to the latest type version format (if it was not converted when the type was altered). You can define the storage for any new column while upgrading the table by using the *column_properties* and the *LOB_partition_storage*. This is the default.

For information on whether a table contains data based on an older type version, refer to the DATA_UPGRADED column of the USER_TAB_COLUMNS data dictionary view.

NOT INCLUDING DATA Specify NOT INCLUDING DATA if you want Oracle to leave column data unchanged.

Restriction on NOT INCLUDING DATA: You cannot specify NOT INCLUDING DATA if the table contains columns in Oracle8 release 8.0.x image format. To

determine whether the table contains such columns, refer to the V80_FMT_IMAGE column of the USER_TAB_COLUMNS data dictionary view.

See Also:

- *Oracle9i Database Reference* for information on the data dictionary views
- ALTER TYPE on page 12-6 for information on converting dependent table data when modifying a type upon which the table depends
- *Oracle9i Application Developer's Guide - Object-Relational Features* for more information on the implications of not converting table data to the latest type version format

records_per_block_clause

The *records_per_block_clause* lets you specify whether Oracle restricts the number of records that can be stored in a block. This clause ensures that any bitmap indexes subsequently created on the table will be as small (compressed) as possible.

Restrictions on the *records_per_block_clause*:

- You cannot specify either MINIMIZE or NOMINIMIZE if a bitmap index has already been defined on table. You must first drop the bitmap index.
- You cannot specify this clause for an index-organized table or nested table.

MINIMIZE Specify MINIMIZE to instruct Oracle to calculate the largest number of records in any block in the table, and limit future inserts so that no block can contain more than that number of records.

Oracle Corporation recommends that a representative set of data already exist in the table before you specify MINIMIZE. If you are using data segment compression (see *data_segment_compression* on page 11-33), then a representative set of *compressed* data should already exist in the table.

Restriction on MINIMIZE: You cannot specify MINIMIZE for an empty table.

NOMINIMIZE Specify NOMINIMIZE to disable the MINIMIZE feature. This is the default.

RENAME TO

Use the RENAME clause to rename *table* to *new_table_name*.

Restriction on the RENAME clause: You cannot rename a materialized view.

> **Note:** Using this clause invalidates any dependent materialized views. For more information on materialized views, see CREATE MATERIALIZED VIEW on page 14-5 and *Oracle9i Data Warehousing Guide*.

row_movement_clause

The *row_movement_clause* lets you specify whether Oracle can move a table row. It is possible for a row to move, for example, during data segment compression or an update operation on partitioned data.

> **Caution:** If you need static rowids for data access, do not enable row movement. For a normal (heap-organized) table, moving a row changes that row's rowid. For a moved row in an index-organized table, the logical rowid remains valid, although the physical guess component of the logical rowid becomes inaccurate.

- Specify ENABLE to allow Oracle to move a row, thus changing the rowid.
- Specify DISABLE if you want to prevent Oracle from moving a row, thus preventing a change of rowid.

Restriction on *row_movement_clause*: You cannot specify this clause for a nonpartitioned index-organized table.

alter_iot_clauses

index_org_table_clause

See *index_org_table_clause* on page 15-31 in the context of CREATE TABLE.

> **See Also:** "Modifying Index-Organized Tables: Examples" on page 11-92

alter_overflow_clause

The *alter_overflow_clause* lets you change the definition of an index-organized table. Index-organized tables keep data sorted on the primary key and are therefore best suited for primary-key-based access and manipulation.

Note: When you add a column to an index-organized table, Oracle evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, then Oracle raises an error and does not execute the ALTER TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.

PCTTHRESHOLD *integer* Specify the percentage of space reserved in the index block for an index-organized table row. PCTTHRESHOLD must be large enough to hold the primary key. All trailing columns of a row, starting with the column that causes the specified threshold to be exceeded, are stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. If you do not specify PCTTHRESHOLD, the default is 50.

Restriction: You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

INCLUDING *column_name* Specify a column at which to divide an index-organized table row into index and overflow portions. The primary key columns are always stored in the index. *column_name* can be either the last primary-key column or any non-primary-key column. All non-primary-key columns that follow *column_name* are stored in the overflow data segment.

Restriction: You cannot specify this clause for individual partitions of an index-organized table.

Note: If an attempt to divide a row at *column_name* causes the size of the index portion of the row to exceed the PCTTHRESHOLD value (either specified or default), Oracle breaks up the row based on the PCTTHRESHOLD value.

---

overflow_attributes The *overflow_attributes* let you specify the overflow data segment physical storage and logging attributes to be modified for the index-organized table. Parameters specified in this clause are applicable only to the overflow data segment.

See Also: CREATE TABLE on page 15-7

add_overflow_clause The *add_overflow_clause* lets you add an overflow data segment to the specified index-organized table. You can also use this clause to explicitly allocate an extent to or deallocate unused space from an existing overflow segment.

Use the STORE IN *tablespace* clause to specify tablespace storage for the entire overflow segment. Use the PARTITION clause to specify tablespace storage for the segment by partition.

For a partitioned index-organized table:

- If you do not specify PARTITION, then Oracle automatically allocates an overflow segment for each partition. The physical attributes of these segments are inherited from the table level.
- If you wish to specify separate physical attributes for one or more partitions, then you must specify such attributes for *every* partition in the table. You need not specify the name of the partitions, but you must specify their attributes in the order in which they were created.

You can find the order of the partitions by querying the PARTITION_NAME and PARTITION_POSITION columns of the USER_IND_PARTITIONS view.

If you do not specify TABLESPACE for a particular partition, then Oracle uses the tablespace specified for the table. If you do not specify TABLESPACE at the table level, then Oracle uses the tablespace of the partition's primary key index segment.

See Also: *allocate_extent_clause* on page 7-2 and *deallocate_unused_clause* on page 7-37 for full descriptions of these clauses of the *add_overflow_clause*

alter_mapping_table_clause

The *alter_mapping_table_clause* is valid only if *table* is index organized and has a mapping table.

UPDATE BLOCK REFERENCES Specify UPDATE BLOCK REFERENCES to update all stale "guess" data block addresses stored as part of the logical ROWID column in the mapping table with the correct address for the corresponding block identified by the primary key.

allocate_extent_clause Use the *allocate_extent_clause* to allocate a new extent at the end of the mapping table for the index-organized table.

> **See Also:** *allocate_extent_clause* on page 7-2 for a full description of this clause

deallocate_unused_clause Specify the *deallocate_unused_clause* to deallocate unused space at the end of the mapping table of the index-organized table.

> **See Also:** *deallocate_unused_clause* on page 7-37 for a full description of this clause

COALESCE

The keyword is relevant only if *table* is index organized. Specify COALESCE to instruct Oracle to combine the primary key index blocks of the index-organized table where possible to free blocks for reuse. You can specify this clause with the *parallel_clause*.

column_clauses

add_column_clause

The *add_column_clause* lets you add a column to a table.

> **See Also:** CREATE TABLE on page 15-7 for a description of the keywords and parameters of this clause and "Adding a Table Column: Example" on page 11-95

If you add a column, then the initial value of each row for the new column is null unless you specify the DEFAULT clause. In this case, Oracle updates each row in the new column with the value you specify for DEFAULT. This update operation, in turn, fires any AFTER UPDATE triggers defined on the table.

Note: If a column has a default value, then you can use the DEFAULT clause to change the default to NULL, but you cannot remove the default value completely. That is, if a column has ever had a default value assigned to it, then the DATA_DEFAULT column of the USER_TAB_COLUMNS data dictionary view will always display either a default value or NULL.

You can add an overflow data segment to each partition of a partitioned index-organized table.

You can add LOB columns to nonpartitioned and partitioned tables. You can specify LOB storage at the table and at the partition or subpartition level.

If you previously created a view with a query that used the "SELECT *" syntax to select all columns from table, and you now add a column to *table*, then Oracle does not automatically add the new column to the view. To add the new column to the view, re-create the view using the CREATE VIEW statement with the OR REPLACE clause.

> **See Also:** CREATE VIEW on page 16-39

Restrictions on Adding Columns:

- You cannot add a LOB column to a clustered table.
- If you add a LOB column to a hash-partitioned table, then the only attribute you can specify for the new partition is TABLESPACE.
- You cannot add a column with a NOT NULL constraint if *table* has any rows unless you also specify the DEFAULT clause.
- If you specify this clause for an index-organized table, then you cannot specify any other clauses in the same statement.

DEFAULT

Use the DEFAULT clause to specify a default for a new column or a new default for an existing column. Oracle assigns this value to the column if a subsequent INSERT statement omits a value for the column. If you are adding a new column to the table and specify the default value, then Oracle inserts the default column value into all rows of the table.

The datatype of the default value must match the datatype specified for the column. The column must also be long enough to hold the default value.

Restrictions on DEFAULT:

- A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.
- The expression can be of any form except a scalar subquery expression.

> **See Also:** "Specifying Default Column Value: Examples" on page 11-96

inline_constraint

Use *inline_constraint* to add a constraint to the new column

inline_ref_constraint

This clause lets you describe a new column of type REF.

> **See Also:** *constraints* on page 7-5 for syntax and description of this type of constraint, including restrictions

column_properties

The *column_properties* determine the storage characteristics of an object, nested table, varray, or LOB column.

object_type_col_properties This clause is valid only when you are adding a new object type column or attribute. To modify the properties of an existing object type column, use the *modify_column_clause*.

Use the *object_type_col_properties* to specify storage characteristics for a new object column or attribute or an element of a collection column or attribute.

column For *column*, specify an object column or attribute.

substitutable_column_clause The *substitutable_column_clause* indicates whether object columns or attributes in the same hierarchy are substitutable for each other. You can specify that a column is of a particular type, or whether it can contain instances of its subtypes, or both.

- If you specify ELEMENT, you constrain the element type of a collection column or attribute to a subtype of its declared type.
- The IS OF [TYPE] (ONLY *type*) clause constrains the type of the object column to a subtype of its declared type.
- NOT SUBSTITUTABLE AT ALL LEVELS indicates that the object column cannot hold instances corresponding to any of its subtypes. Also, substitution is disabled for any embedded object attributes and elements of embedded nested tables and varrays. The default is SUBSTITUTABLE AT ALL LEVELS.

  Restrictions on the *substitutable_column_clause*:

  - You cannot specify this clause for an attribute of an object column. However, you can specify this clause for a object type column of a relational

table, and for an object column of an object table if the substitutability of the object table itself has not been set.

- For a collection type column, the only part of this clause you can specify is `[NOT] SUBSTITUTABLE AT ALL LEVELS`.

nested_table_col_properties The *nested_table_col_properties* clause lets you specify separate storage characteristics for a nested table, which in turn lets you to define the nested table as an index-organized table. You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here.)

- For *nested_item*, specify the name of a column (or a top-level attribute of the table's object type) whose type is a nested table.

  If the nested table is a multilevel collection, then the inner nested table may not have a name. In this case, specify `COLUMN_VALUE` in place of the *nested_item* name.

- For *storage_table*, specify the name of the table where the rows of *nested_item* reside. The storage table is created in the same schema and the same tablespace as the parent table.

Restrictions on *nested_table_column_properties*:

- You cannot specify the *parallel_clause*.
- You cannot specify `TABLESPACE` (as part of the *segment_attributes_clause*) for a nested table. The tablespace is always that of the parent table.
- You cannot specify `CLUSTER` as part of the *physical_properties* clause.

See Also: "Nested Tables: Examples" on page 11-98

varray_col_properties The *varray_col_properties* clause lets you specify separate storage characteristics for the LOB in which a varray will be stored. If you specify this clause, then Oracle will always store the varray in a LOB, even if it is small enough to be stored inline. If *varray_item* is a multilevel collection, then Oracle stores all collection items nested within *varray_item* in the same LOB in which *varray_item* is stored.

Restriction on *varray_col_properties*: You cannot specify `TABLESPACE` as part of *LOB_parameters* for a varray column. The LOB tablespace for a varray defaults to the containing table's tablespace.

LOB_storage_clause Use the *LOB_storage_clause* to specify the LOB storage characteristics for a newly added LOB column, partition, or subpartition. You cannot use this clause to modify an existing LOB. Instead, you must use the *modify_LOB_storage_clause*.

CACHE READS Clause CACHE READS applies only to LOB storage. It indicates that LOB values are brought into the buffer cache only during read operations, but not during write operations.

- For *LOB_item*, specify the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table.
- For *LOB_segname*, specify the name of the LOB data segment. You cannot use *LOB_segname* if more than one *LOB_item* is specified.

When you add a new LOB column, you can specify the logging attribute with CACHE READS, as you can when defining a LOB column at create time.

When you modify a LOB column from CACHE or NOCACHE to CACHE READS, or from CACHE READS to CACHE or NOCACHE, you can change the logging attribute. If you do not specify LOGGING or NOLOGGING, then this attribute defaults to the current logging attribute of the LOB column.

For existing LOBs, if you do not specify CACHE, NOCACHE, or CACHE READS, then Oracle retains the existing values of the LOB attributes.

Restrictions on *LOB_parameters*:

- The only parameter of *LOB_parameters* you can specify for a hash partition or hash subpartition is TABLESPACE.
- You cannot specify the *LOB_index_clause* if *table* is partitioned.

ENABLE | DISABLE STORAGE IN ROW Specify whether the LOB value is to be stored in the row (inline) or outside of the row (out of line). (The LOB locator is always stored inline regardless of where the LOB value is stored.)

- ENABLE specifies that the LOB value is stored inline if its length is less than approximately 4000 bytes minus system control information. This is the default.
- DISABLE specifies that the LOB value is stored out of line regardless of the length of the LOB value.

Restrictions on enabling storage in row: You cannot change STORAGE IN ROW once it is set. Therefore, you cannot specify this clause as part of the *modify_column_*

options clause. However, you can change this setting when adding a new column (*add_column_clause*) or when moving the table (*move_table_clause*).

CHUNK *integer* Specify the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, then Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, then Oracle allocates 4096 bytes (2 blocks).The maximum value is 32768 (32 K), which is the largest Oracle block size allowed. The default CHUNK size is one Oracle database block.

Restrictions on CHUNK:

- You cannot change the value of CHUNK once it is set.
- The value of CHUNK must be less than or equal to the value of NEXT (either the default value or that specified in the storage clause). If CHUNK exceeds the value of NEXT, then Oracle returns an error.

PCTVERSION *integer* Specify the maximum percentage of overall LOB storage space to be used for maintaining old versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until 10% of the overall LOB storage space is used.

RETENTION If the database is in automatic undo mode, then you can specify RETENTION instead of PCTVERSION to instruct Oracle to retain old versions of this LOB. This clause overrides any prior setting of PCTVERSION.

Restriction on RETENTION: You cannot specify RETENTION if the database is running in manual undo mode.

> **See Also:** *LOB_parameters* on page 15-37 for a full description of the RETENTION parameter

FREEPOOLS *integer* If the database is in automatic undo mode, then you can use this clause to specify the number of freelist groups for this LOB. This clause overrides any prior setting of FREELIST GROUPS.

Restriction on FREEPOOLS: You cannot specify FREEPOOLS if the database is running in manual undo mode.

> **See Also:** *LOB_parameters* on page 15-37 for a full description of the FREEPOOLS parameter

LOB_index_clause This clause has been deprecated since Oracle8*i*. Oracle generates an index for each LOB column. The LOB indexes are system named and system managed, and they reside in the same tablespace as the LOB data segments.

It is still possible for you to specify this clause in some cases. However, Oracle Corporation strongly recommends that you no longer do so. In any event, do not put the LOB index in a different tablespace from the LOB data.

> **See Also:** *Oracle9i Database Migration* for information on how Oracle manages LOB indexes in tables migrated from earlier versions

LOB_partition_storage

The *LOB_partition_storage* clause lets you specify a separate *LOB_storage_clause* or *varray_col_properties* clause for each partition. You must specify the partitions in the order of partition position. You can find the order of the partitions by querying the PARTITION_NAME and PARTITION_POSITION columns of the USER_IND_PARTITIONS view.

If you do not specify a *LOB_storage_clause* or *varray_col_properties* clause for a particular partition, then the storage characteristics are those specified for the LOB item at the table level. If you also did not specify any storage characteristics for the LOB item at the table level, then Oracle stores the LOB data partition in the same tablespace as the table partition to which it corresponds.

Restriction on *LOB_partition_storage***:** You can specify only one list of *LOB_partition_storage* clause in a single ALTER TABLE statement, and all *LOB_storage_clauses* and *varray_col_properties* clause must precede the list of *LOB_partition_storage* clauses.

XMLType_column_properties The *XMLType_column_properties* let you specify storage attributes for an XMLTYPE column.

XMLType_storage XMLType columns can be stored either in LOB or object-relational columns.

- Specify STORE AS OBJECT RELATIONAL if you want Oracle to store the XMLType data in object-relational columns. Storing data object relationally lets you define indexes on the relational columns and enhances query performance.

  If you specify object-relational storage, you must also specify the *XMLSchema_spec* clause.

- Specify STORE AS CLOB if you want Oracle to store the XMLType data in a CLOB column. Storing data in a CLOB column preserves the original content and enhances retrieval time.

  If you specify LOB storage, you can specify either LOB parameters or the *XMLSchema_spec* clause, but not both. Specify the *XMLSchema_spec* clause if you want to restrict the table or column to particular schema-based XML instances.

XMLSchema_spec This clause lets you specify the URL of a registered XMLSchema (in the XMLSCHEMA clause or as part of the ELEMENT clause) and an XML element name. You must specify an element, although the XMLSchema URL is optional. If you do specify an XMLSchema URL, you must already have registered the XMLSchema using the DBMS_XMLSCHEMA package.

See Also:

- *LOB_storage_clause* on page 11-45 for information on the *LOB_segname* and *LOB_parameters* clauses
- "XMLType Column Examples" on page 15-72 for an example of XMLType columns in object-relational tables and "Using XML in SQL Statements" on page D-11 for an example of creating an XMLSchema
- *Oracle9i XML Database Developer's Guide - Oracle XML DB* for more information on XMLType columns and tables and on creating XMLSchemas

modify_column_clause

Use the *modify_column_clause* to modify the properties of an existing column or the substitutability of an existing object type column.

See Also: "Modifying Table Columns: Examples" on page 11-95

modify_col_properties

Use this clause to modify the properties of the column. Any of the optional parts of the column definition (datatype, default value, or constraint) that you omit from this clause remain unchanged.

datatype You can change any column's datatype if all rows for the column contain nulls. However, if you change the datatype of a column in a materialized view container table, then the corresponding materialized view is invalidated.

You can omit the datatype only if the statement also designates the column as part of the foreign key of a referential integrity constraint. Oracle automatically assigns the column the same datatype as the corresponding column of the referenced key of the referential integrity constraint.

You can always increase the size of a character or raw column or the precision of a numeric column, whether or not all the columns contain nulls. You can reduce the size of a column's datatype as long as the change does not require data to be modified. Oracle scans existing data and returns an error if data exists that exceeds the new length limit.

You can modify a DATE column to TIMESTAMP or TIMESTAMP WITH LOCAL TIME ZONE. You can modify any TIMESTAMP WITH LOCAL TIME ZONE to a DATE column.

Note: When you modify a TIMESTAMP WITH LOCAL TIME ZONE column to a DATE column, the fractional seconds and time zone adjustment data is lost.

- If the TIMESTAMP WITH LOCAL TIME ZONE data has fractional seconds, then Oracle updates the row data for the column by rounding the fractional seconds.
- If the TIMESTAMP WITH LOCAL TIME ZONE data has the minute field greater than equal to 60 (which can occur in a boundary case when the daylight savings rule switches), then Oracle updates the row data for the column by subtracting 60 from its minute field.

If the table is empty, then you can increase or decrease the leading field or the fractional second value of a datetime or interval column. If the table is not empty, then you can only increase the leading field or fractional second of a datetime or interval column.

You can change a LONG column to a CLOB or NCLOB column, and a LONG RAW column to a BLOB column.

- The modified LOB column inherits all constraints and triggers that were defined on the original LONG column. If you wish to change any constraints, then you must do so in a subsequent ALTER TABLE statement.

- If any domain indexes are defined on the LONG column, then you must drop them before modifying the column to a LOB.
- After the modification, you will have to rebuild all other indexes on all columns of the table.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information on LONG and LOB columns
- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for information on LONG to LOB migration
- ALTER INDEX on page 9-64 for information on dropping and rebuilding indexes
- "Converting LONG Columns to LOB: Example" on page 11-96

For CHAR and VARCHAR2 columns, you can change the length semantics by specifying CHAR (to indicate character semantics for a column that was originally specified in bytes) or BYTE (to indicate byte semantics for a column that was originally specified in characters). To learn the length semantics of existing columns, query the CHAR_USED column of the ALL_, USER_, or DBA_TAB_COLUMNS data dictionary view.

See Also:

- *Oracle9i Database Globalization Support Guide* for information on byte and character semantics
- *Oracle9i Database Reference* for information on the data dictionary views

inline_constraint The only type of integrity constraint that you can add to an existing column using the MODIFY clause is a NOT NULL constraint, and only if the column contains no nulls. To define other types of integrity constraints (UNIQUE, PRIMARY KEY, referential integrity, and CHECK constraints) on existing columns, use the *add_column_clause*. To modify existing constraints on existing columns, use the *constraint_clauses*.

Restrictions on *modify_col_properties*:

- You cannot modify a column of a table if a domain index is defined on the column. You must first drop the domain index and then modify the column.

- You cannot specify a column of datatype ROWID for an index-organized table, but you can specify a column of type UROWID.
- You cannot change a column's datatype to REF.

> **See Also:** ALTER MATERIALIZED VIEW on page 9-92 for information on revalidating a materialized view

modify_col_substitutable

Use this clause to set or change the substitutability of an existing object type column.

The FORCE keyword drops any hidden columns containing typeid information or data for subtype attributes. You must specify FORCE if the column or any attributes of its type are not FINAL.

Restrictions on *modify_col_substitutable*:

- You can specify this clause only once in any ALTER TABLE statement.
- You cannot modify the substitutability of a column in an object table if the substitutability of the table itself has been set.
- You cannot specify this clause if the column was created or added using the IS OF TYPE syntax (see *substitutable_column_clause* on page 11-43), which limits the range of subtypes permitted in an object column or attribute to a particular subtype.
- You cannot change a varray column to NOT SUBSTITUTABLE if any of its attributes of nested object types is not FINAL, even by specifying FORCE.

drop_column_clause

The *drop_column_clause* lets you free space in the database by dropping columns you no longer need, or by marking them to be dropped at a future time when the demand on system resources is less.

- If you drop a nested table column, then its storage table is removed.
- If you drop a LOB column, then the LOB data and its corresponding LOB index segment are removed.
- If you drop a BFILE column, then only the locators stored in that column are removed, not the files referenced by the locators.

- If you drop (or mark unused) a column defined as an INCLUDING column, then the column stored immediately before this column will become the new INCLUDING column.

SET UNUSED Clause

Specify SET UNUSED to mark one or more columns as unused. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than it would be if you execute the DROP clause.

You can view all tables with columns marked UNUSED in the data dictionary views USER_UNUSED_COL_TABS, DBA_UNUSED_COL_TABS, and ALL_UNUSED_COL_TABS.

> **See Also:** *Oracle9i Database Reference* for information on the data dictionary views

Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked UNUSED, you have no access to that column. A "SELECT *" query will not retrieve data from unused columns. In addition, the names and types of columns marked UNUSED will not be displayed during a DESCRIBE, and you can add to the table a new column with the same name as an unused column.

Note: Until you actually drop these columns, they continue to count toward the absolute limit of 1000 columns in a single table. However, as with all DDL statements, you cannot roll back the results of this clause. That is, you cannot issue SET USED counterpart to retrieve a column that you have SET UNUSED.

Also, if you mark a column of datatype LONG as UNUSED, then you cannot add another LONG column to the table until you actually drop the unused LONG column.

> **See Also:** CREATE TABLE on page 15-7 for more information on the 1000-column limit

DROP Clause

Specify DROP to remove the column descriptor and the data associated with the target column from each row in the table. If you explicitly drop a particular column,

then all columns currently marked UNUSED in the target table are dropped at the same time.

When the column data is dropped:

- All indexes defined on any of the target columns are also dropped.
- All constraints that reference a target column are removed.
- If any statistics types are associated with the target columns, then Oracle disassociates the statistics from the column with the FORCE option and drops any statistics collected using the statistics type.

Note: If the target column is a parent key of a nontarget column, or if a check constraint references both the target and nontarget columns, then Oracle returns an error and does not drop the column unless you have specified the CASCADE CONSTRAINTS clause. If you have specified that clause, then Oracle removes all constraints that reference any of the target columns.

See Also: DISASSOCIATE STATISTICS on page 16-64 for more information on disassociating statistics types

DROP UNUSED COLUMNS Clause

Specify DROP UNUSED COLUMNS to remove from the table all columns currently marked as unused. Use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, then the statement returns with no errors.

column Specify one or more columns to be set as unused or dropped. Use the COLUMN keyword only if you are specifying only one column. If you specify a column list, then it cannot contain duplicates.

CASCADE CONSTRAINTS Specify CASCADE CONSTRAINTS if you want to drop all foreign key constraints that refer to the primary and unique keys defined on the dropped columns, and drop all multicolumn constraints defined on the dropped columns. If any constraint is referenced by columns from other tables or remaining columns in the target table, then you must specify CASCADE CONSTRAINTS. Otherwise, the statement aborts and an error is returned.

INVALIDATE The INVALIDATE keyword is optional. Oracle automatically invalidates all dependent objects, such as views, triggers, and stored program units. Object invalidation is a recursive process. Therefore, all directly dependent and indirectly dependent objects are invalidated. However, only local dependencies are invalidated, because Oracle manages remote dependencies differently from local dependencies.

An object invalidated by this statement is automatically revalidated when next referenced. You must then correct any errors that exist in that object before referencing it.

> **See Also:** *Oracle9i Database Concepts* for more information on dependencies

CHECKPOINT Specify CHECKPOINT if you want Oracle to apply a checkpoint for the DROP COLUMN operation after processing *integer* rows; *integer* is optional and must be greater than zero. If *integer* is greater than the number of rows in the table, then Oracle applies a checkpoint after all the rows have been processed. If you do not specify *integer*, then Oracle sets the default of 512. Checkpointing cuts down the amount of undo logs accumulated during the DROP COLUMN operation to avoid running out of rollback segment space. However, if this statement is interrupted after a checkpoint has been applied, then the table remains in an unusable state. While the table is unusable, the only operations allowed on it are DROP TABLE, TRUNCATE TABLE, and ALTER TABLE DROP COLUMNS CONTINUE (described in sections that follow).

You cannot use this clause with SET UNUSED, because that clause does not remove column data.

DROP COLUMNS CONTINUE Clause

Specify DROP COLUMNS CONTINUE to continue the drop column operation from the point at which it was interrupted. Submitting this statement while the table is in a valid state results in an error.

Restrictions on the *drop_column_clause:*

- Each of the parts of this clause can be specified only once in the statement and cannot be mixed with any other ALTER TABLE clauses. For example, the following statements are not allowed:

```
ALTER TABLE t1 DROP COLUMN f1 DROP (f2);
ALTER TABLE t1 DROP COLUMN f1 SET UNUSED (f2);
ALTER TABLE t1 DROP (f1) ADD (f2 NUMBER);
ALTER TABLE t1 SET UNUSED (f3)
```

```
ADD (CONSTRAINT ck1 CHECK (f2 > 0));
```

- You can drop an object type column only as an entity. To drop an attribute from an object type column, use the ALTER TYPE ... DROP ATTRIBUTE statement with the CASCADE INCLUDING TABLE DATA clause. Be aware that dropping an attribute affects all dependent objects. See DROP ATTRIBUTE on page 12-15 for more information.
- You can drop a column from an index-organized table only if it is not a primary key column. The primary key constraint of an index-organized table can never be dropped, so you cannot drop a primary key column even if you have specified CASCADE CONSTRAINTS.
- You can export tables with dropped or unused columns. However, you can import a table only if all the columns specified in the export files are present in the table (that is, none of those columns has been dropped or marked unused). Otherwise, Oracle returns an error.
- You cannot drop a column on which a domain index has been built.
- You cannot drop a SCOPE table constraint or a WITH ROWID constraint on a REF column.
- You cannot use this clause to drop:
  - A pseudocolumn, cluster column, or partitioning column. (You can drop nonpartitioning columns from a partitioned table if all the tablespaces where the partitions were created are online and in read / write mode.)
  - A column from: a nested table, an object table, or a table owned by SYS.

See Also: "Dropping a Column: Example" on page 11-91

rename_column_clause

Use the *rename_column_clause* to rename a column of *table*. The new column name must not be the same as any other column name in *table*.

When you rename a column, Oracle handles dependent objects as follows:

- Function-based indexes and check constraints that depend on the renamed column remain valid.
- Dependent views, triggers, domain indexes, functions, procedures, and packages are marked INVALID. Oracle attempts to revalidate them when they are next accessed, but you may need to alter these objects with the new column name if revalidation fails.

Restrictions on the *rename_column_clause*:

- You cannot combine this clause with any of the other *column_clauses* in the same statement.
- You cannot rename a column that is used to define a join index. Instead you must drop the index, rename the column, and re-create the index.

See Also: "Renaming a Column: Example" on page 11-91

modify_collection_retrieval

Use the *modify_collection_retrieval* clause to change what Oracle returns when a collection item is retrieved from the database.

collection_item Specify the name of a column-qualified attribute whose type is nested table or varray.

RETURN AS Specify what Oracle should return as the result of a query:

- LOCATOR specifies that a unique locator for the nested table is returned.
- VALUE specifies that a copy of the nested table itself is returned.

See Also: "Collection Retrieval: Example" on page 11-88

modify_LOB_storage_clause

The *modify_LOB_storage_clause* lets you change the physical attributes of *LOB_item*. You can specify only one *LOB_item* for each *modify_LOB_storage_clause*.

The REBUILD FREEPOOLS clause removes all the old data from the LOB column. This clause is useful only if you reverting to PCTVERSION for management of LOBs. You might want to do this to manage older data blocks, and you must do this if you are downgrading to a release of Oracle earlier than 9.2.0.

Restrictions on *modify_LOB_storage_clause*:

- You cannot modify the value of the INITIAL parameter in the *storage_clause* when modifying the LOB storage attributes.
- You cannot specify both the *allocate_extent_clause* and the *deallocate_unused_clause* in the same statement.
- You cannot specify both.

> **See Also:** *LOB_storage_clause* of CREATE TABLE on page 15-37 for information on setting LOB parameters and "LOB Columns: Examples" on page 11-98

alter_varray_col_properties

The *alter_varray_col_properties* clause lets you change the storage characteristics of an existing LOB in which a varray is stored.

Restriction on *alter_varray_col_properties*: You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the tablespace of the containing table.

constraint_clauses

Use the *constraint_clauses* to add a new constraint using out-of-line declaration, modify the state of an existing constraint, or to drop a constraint.

> **See Also:** *constraints* on page 7-5 for a description of all the keywords and parameters of out-of-line constraints and *constraint_state*

Adding a Constraint

The ADD clause lets you add a new out-of-line constraint or out-of-line REF constraint to the table.

> **See Also:** "Disabling a CHECK Constraint: Example" on page 11-90, "Specifying Object Identifiers: Example" on page 11-95, and "REF Columns: Examples" on page 11-99

Modifying a Constraint

The MODIFY CONSTRAINT clause lets you change the state of an existing constraint.

Restrictions on modifying constraints:

- You cannot modify the datatype or length of a column that is part of a table or index partitioning or subpartitioning key.
- You can change a CHAR column to VARCHAR2 (or VARCHAR) and a VARCHAR2 (or VARCHAR) to CHAR only if the column contains nulls in all rows or if you do not attempt to change the column size.
- You cannot change a LONG or LONG RAW column to a LOB if it is part of a cluster. If you do change a LONG or LONG RAW column to a LOB, then the only

other clauses you can specify in this ALTER TABLE statement are the DEFAULT clause and the *LOB_storage_clause*.

- You can specify the *LOB_storage_clause* as part of *modify_column_options* only when you are changing a LONG or LONG RAW column to a LOB.
- If you specify this clause for an index-organized table, then you cannot specify any other clauses in the same statement.

> **See Also:** "Changing the State of a Constraint: Examples" on page 11-89

Renaming a Constraint

The RENAME CONSTRAINT clause lets you rename any existing constraint on *table*. The new constraint name cannot be the same as any existing constraint on any object in the same schema. All objects that are dependent on the constraint remain valid.

> **See Also:** "Renaming Constraints: Example" on page 11-97

drop_constraint_clause

The *drop_constraint_clause* lets you drop an integrity constraint from the database. Oracle stops enforcing the constraint and removes it from the data dictionary. You can specify only one constraint for each *drop_constraint_clause*, but you can specify multiple *drop_constraint_clauses* in one statement.

PRIMARY KEY Specify PRIMARY KEY to drop the table's primary key constraint.

UNIQUE Specify UNIQUE to drop the unique constraint on the specified columns.

Note: If you drop the primary key or unique constraint from a column on which a bitmap join index is defined, then Oracle invalidates the index. See CREATE INDEX on page 13-62 for information on bitmap join indexes.

CONSTRAINT Specify CONSTRAINT *constraint* to drop an integrity constraint other than a primary key or unique constraint.

CASCADE Specify CASCADE if you want all other integrity constraints that depend on the dropped integrity constraint to be dropped as well.

KEEP | DROP INDEX Specify KEEP or DROP INDEX to indicate whether Oracle should preserve or drop the index it has been using to enforce the PRIMARY KEY or UNIQUE constraint.

Restrictions on the *drop_constraint_clause*:

- You cannot drop a primary key or unique key constraint that is part of a referential integrity constraint without also dropping the foreign key. To drop the referenced key and the foreign key together, use the CASCADE clause. If you omit CASCADE, then Oracle does not drop the primary key or unique constraint if any foreign key references it.
- You cannot drop a primary key constraint (even with the CASCADE clause) on a table that uses the primary key as its object identifier (OID).
- If you drop a referential integrity constraint on a REF column, then the REF column remains scoped to the referenced table.
- You cannot drop the scope of the column.

See Also: "Dropping Constraints: Examples" on page 11-97

alter_external_table_clause

Use the *alter_external_table_clause* to change the characteristics of an external table. This clause has no affect on the external data itself. The syntax and semantics of the *parallel_clause*, *enable_disable_clause*, *external_data_properties*, and REJECT LIMIT clause are the same as described for CREATE TABLE. See the *external_table_clause* of CREATE TABLE *on page 15-33*.

Restrictions on altering external tables:

- You cannot modify an external table using any clause outside of this clause.
- You cannot add a LONG, LOB, or object type column to an external table, nor can you change the datatype of an external table column to any of these datatypes.
- You cannot add a constraint to an external table.
- You cannot modify the storage parameters of an external table.

alter_table_partitioning

The clauses in this section apply only to partitioned tables. You cannot combine partition operations with other partition operations or with operations on the base table in the same ALTER TABLE statement.

Notes on altering table partitioning:

- The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle9i Database Administrator's Guide* for a discussion of these restrictions.
- For additional information on partition operations on tables with an associated CONTEXT domain index, please refer to *Oracle Text Reference*.
- If you drop, exchange, truncate, move, modify, or split a partition on a table that is a master table for one or more materialized views, then existing bulk load information about the table will be deleted. Therefore, be sure to refresh all dependent materialized views before performing any of these operations.
- If a bitmap join index is defined on *table*, then any operation that alters a partition of *table* causes Oracle to mark the index UNUSABLE.

modify_table_default_attrs

The *modify_table_default_attrs* clause lets you specify new default values for the attributes of *table*. Partitions and LOB partitions you create subsequently will inherit these values unless you override them explicitly when creating the partition or LOB partition. Existing partitions and LOB partitions are not affected by this clause.

Only attributes named in the statement are affected, and the default values specified are overridden by any attributes specified at the individual partition level.

- FOR PARTITION applies only to composite-partitioned tables. This clause specifies new default values for the attributes of *partition*. Subpartitions and LOB subpartitions of *partition* that you create subsequently will inherit these values unless you override them explicitly when creating the subpartition or LOB subpartition. Existing subpartitions are not affected by this clause.
- PCTTHRESHOLD, *key_compression*, and the *alter_overflow_clause* are valid only for partitioned index-organized tables. However, in the *key_compression* clause, you cannot specify an integer after the COMPRESS keyword. Key compression length can be specified only when you create the table.
- You cannot specify the PCTUSED parameter in *segment_attributes* for the index segment of an index-organized table.
- You can specify the *key_compression_clause* only if key compression is already specified at the table level.

set_subpartition_template

Use the *set_subpartition_template* clause to create or replace existing default list or hash subpartition definitions for each table partition. This clause is valid only for composite-partitioned tables. It replaces the existing subpartition template or creates a new template if you have not previously created one. Existing subpartitions are not affected, nor are existing local and global indexes. However, subsequent partitioning operations (such as add and merge operations) will use the new template.

You can drop an existing subpartition template by specifying ALTER TABLE *table* SET SUBPARTITION TEMPLATE ( ).

Restrictions on *set_subpartition_template*:

- For a range-hash composite-partitioned table, you cannot specify the *list_values_clause*.
- For a range-list composite-partitioned table, you cannot specify the *hash_subpartition_quantity* clause.
- For both range-hash and range-list partitioned tables, the only clause of the *partitioning_storage_clause* you can specify for subpartitions is the TABLESPACE clause.

modify_table_partition

The *modify_table_partition* clause lets you change the real physical attributes of a range, hash, or list partition. This clause optionally modifies the storage attributes of one or more LOB items for the partition. You can specify new values for physical attributes (with some restrictions, as noted in the sections that follow), logging; and storage parameters.

You can also specify how Oracle should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses" on page 11-83.

For partitioned index-organized tables, you can also update the mapping table in conjunction with partition changes. See the *alter_mapping_table_clause* on page 11-40.

See Also: "Modifying Table Partitions: Examples" on page 11-94

modify_range_partition

When modifying a range partition, if *table* is composite partitioned:

- If you specify the *allocate_extent_clause*, then Oracle allocates an extent for each subpartition of *partition*.
- If you specify *deallocate_unused_clause*, then Oracle deallocates unused storage from each subpartition of *partition*.
- Any other attributes changed in this clause will be changed in subpartitions of *partition* as well, overriding existing values. To avoid changing the attributes of existing subpartitions, use the FOR PARTITION clause of *modify_table_default_attrs*.

Restriction on *modify_range_partition*: If you specify UNUSABLE LOCAL INDEXES, then you cannot specify any other clause of *modify_range_partition*.

add_hash_subpartition This clause is valid only for range-hash composite partitions. The *add_hash_subpartition* clause lets you add a hash subpartition to *partition*. Oracle populates the new subpartition with rows rehashed from the other subpartition(s) of *partition* as determined by the hash function. For optimal load balancing, the total number of subpartitions should be a power of 2.

- If you do not specify *subpartition*, then Oracle assigns a name in the form SYS_SUBP*n*.
- The *list_values_clause* is not valid for this operation.
- In the *partitioning_storage_clause*, the only clause you can specify for subpartitions is the TABLESPACE clause. If you do not specify TABLESPACE, then the new subpartition will reside in the default tablespace of *partition*.

Oracle invalidates any global indexes on *table*. You can update these indexes during this operation using the *update_global_index_clause*.

Oracle adds local index partitions corresponding to the selected partition. Oracle marks UNUSABLE, and you must rebuild, the local index partitions corresponding to the added partitions.

add_list_subpartition the *add_list_subpartition* clause lets you add a list subpartition to partition. This clause is valid only for range-list composite partitions, and only if you have not already created a DEFAULT subpartition.

- If you do not specify *subpartition*, then Oracle assigns a name in the form SYS_SUBP*n*.
- The *list_values_clause* is required in this operation, and the values you specify in the *list_values_clause* cannot exist in any other subpartition of

partition. However, these values can duplicate values found in subpartitions of other partitions.

- In the *partitioning_storage_clause*, the only clause you can specify for subpartitions is the TABLESPACE clause. If you do not specify TABLESPACE, then Oracle stores the new subpartition in the default tablespace of *partition*. If partition has no default tablespace, then Oracle uses the default tablespace of table. If table has no default tablespace, then Oracle uses the default tablespace of the user.

Oracle also adds a subpartition with the same value list to all local index partitions of the table. The status of existing local and global index partitions of table are not affected.

Restriction on *add_list_subpartition*: You cannot specify this clause if you have already created a DEFAULT subpartition for this partition. Instead you must split the DEFAULT partition using the *split_list_subpartition* clause.

modify_hash_partition

When modifying a hash partition, in the *partition_attributes* clause, you can specify only the *allocate_extent_clause* and *deallocate_unused_clause*. All other attributes of the partition are inherited from the table-level defaults except TABLESPACE, which stays the same as it was at create time.

COALESCE SUBPARTITION COALESCE SUBPARTITION applies only to hash subpartitions. Use the COALESCE SUBPARTITION clause if you want Oracle to select the last hash subpartition, distribute its contents into one or more remaining subpartitions (determined by the hash function), and then drop the last subpartition.

Oracle invalidates any global indexes on *table*. You can update these indexes during this operation using the *update_global_index_clause*.

Oracle drops local index partitions corresponding to the selected partition. Oracle marks UNUSABLE, and you must rebuild, the local index partitions corresponding to one or more absorbing partitions.

Restriction on *modify_hash_partition*: If you specify UNUSABLE LOCAL INDEXES, then you cannot specify any other clause of *modify_hash_partition*.

modify_list_partition

When modifying a list partition, the following additional clauses are available:

ADD | DROP VALUES Clauses These clauses are valid only when you are modifying list partitions. Local and global indexes on the table are not affected by either of these clauses.

- Use the ADD VALUES clause to extend the *partition_value* list of *partition* to include additional values. The added partition values must comply with all rules and restrictions listed in the *list_partitioning* of CREATE TABLE on page 15-48.
- Use the DROP VALUES clause to reduce the *partition_value* list of *partition* by eliminating one or more *partition_value*. When you specify this clause, Oracle checks to ensure that no rows with this value exist. If such rows do exist, then Oracle returns an error.

> **Note:** An ADD VALUES operation on a table with a DEFAULT list partition will be enhanced if you have defined a local prefixed index on the table. A DROP VALUES operation also will be enhanced by such an index.

Restrictions on adding and dropping list values:

- You cannot add values to a default list partition. If *table* contains a default partition and you attempt to add values to a nondefault partition, then Oracle will check that the values being added do not already exist in the default partition. If the values do exist in the default partition, then Oracle returns an error.
- You cannot drop values from a default partition.

Restriction on *modify_list_partition*: If you specify UNUSABLE LOCAL INDEXES, then you cannot specify any other clause of *modify_list_partition*.

modify_table_subpartition

This clause applies only to composite-partitioned tables.

modify_hash_subpartition

The *modify_hash_subpartition* clause lets you allocate or deallocate storage for an individual subpartition of *table*. This clause is valid only for range-hash composite-partitioned tables.

You can also specify how Oracle should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses" on page 11-83.

Restriction on *modify_hash_subpartition*: The only *modify_LOB_parameters* you can specify for *subpartition* are the *allocate_extent_clause* and *deallocate_unused_clause*.

modify_list_subpartition

The *modify_list_subpartition* clause lets you make the same changes to a list subpartition that you can make to a hash subpartition. In addition, it lets you add or remove values from a list subpartition's value list. This clause is valid only for range-list composite-partitioned tables.

ADD VALUES Specify ADD VALUES to extend the value list of *subpartition*.

- The values you specify cannot already exist in the value list of *subpartition* or of any other subpartition of the same partition. However, the values can exist in the value lists of subpartitions of other partitions.
- If you have defined a DEFAULT subpartition, then Oracle verifies that none of the values you are adding exist in rows of the DEFAULT subpartition. If the added values do exist in the DEFAULT subpartition, then the statement will fail.
- Oracle adds corresponding values to the value list of any local index subpartitions. The status of local and global index partitions is not affected by this operation.

DROP VALUES Specify DROP VALUES to remove one or more values from the value list of *subpartition*.

- The values you specify must be a subset of existing values in *subpartition*.
- You cannot use this clause to drop all values in a subpartition. Instead you must use an ALTER TABLE ... DROP SUBPARTITION statement.
- If *subpartition* contains any rows containing one of the values being dropped, then the operation fails and Oracle returns an error. You must first delete any rows containing the values you wish to drop before reissuing the statement.
- Oracle updates the value list of any corresponding local index subpartition. The status of local and global index partitions is not affected by this operation.

You can also specify how Oracle should handle local indexes that become unusable as a result of the modification to the partition. See "UNUSABLE LOCAL INDEXES Clauses" on page 11-83.

Restriction on *modify_list_subpartition*: The only *modify_LOB_parameters* you can specify for *subpartition* are the *allocate_extent_clause* and *deallocate_unused_clause*.

move_table_partition

Use the *move_table_partition* clause to move *partition* to another segment. You can move partition data to another tablespace, recluster data to reduce fragmentation, or change create-time physical attributes.

If the table contains LOB columns, then you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this partition. Only the LOBs named are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, then its LOB data and LOB index segments are not moved.

Oracle invalidates any global indexes on heap-organized tables. You can update these indexes during this operation using the *update_global_index_clause*. Global indexes on index-organized tables are primary key based, so they do not become unusable.

Oracle moves local index partitions corresponding to the specified partition. If the moved partitions are not empty, then Oracle marks them UNUSABLE, and you must rebuild them.

When you move a LOB data segment, Oracle drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

The move operation obtains its parallel attribute from the *parallel_clause*, if specified. If not specified, the default parallel attributes of the table, if any, are used. If neither is specified, then Oracle performs the move without using parallelism.

Specifying the *parallel_clause* in MOVE PARTITION does not change the default parallel attributes of *table*.

Note: For index-organized tables, Oracle uses the address of the primary key, as well as its value, to construct logical rowids. The logical rowids are stored in the secondary index of the table. If you move a partition of an index-organized table, then the address portion of the rowids will change, which can hamper performance. To ensure optimal performance, rebuild the secondary index(es) on the moved partition to update the rowids.

See Also: *Oracle9i Database Concepts* for more information on logical rowids and "Moving Table Partitions: Example" on page 11-94

MAPPING TABLE The MAPPING TABLE clause is relevant only for an index-organized table that already has a mapping table defined for it. Oracle moves the mapping table along with the index partition and marks all corresponding bitmap index partitions UNUSABLE.

See Also: *mapping_table_clause* of CREATE TABLE on page 15-32

Restrictions on moving table partitions:

- If *partition* is a hash partition, then the only attribute you can specify in this clause is TABLESPACE.
- You cannot specify this clause for a partition containing subpartitions. However, you can move subpartitions using the *move_table_subpartition_clause*.

move_table_subpartition

Use the *move_table_subpartition* clause to move *subpartition* to another segment. If you do not specify TABLESPACE, then the subpartition remains in the same tablespace.

You can update global indexes on *table* during this operation using the *update_global_index_clause*. If the subpartition is not empty, then Oracle marks UNUSABLE, and you must rebuild, all local index subpartitions corresponding to the subpartition being moved.

If the table contains LOB columns, then you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this subpartition.

Only the LOBs specified are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, then its LOB data and LOB index segments are not moved.

When you move a LOB data segment, Oracle drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

Restriction on moving table subpartitions: In *subpartition_spec*, the only clause of the *partitioning_storage_clause* you can specify is the TABLESPACE clause.

add_table_partition

Use the *add_table_partition* clause to add a hash, range, or list partition to *table*.

Oracle adds to any local index defined on *table* a new partition with the same name as that of the base table partition. If the index already has a partition with such a name, then Oracle generates a partition name of the form SYS_P*n*.

If *table* is index organized, then Oracle adds a partition to any mapping table and overflow area defined on the table as well.

> **See Also:** "Adding a Table Partition with a LOB: Examples" on page 11-93

add_range_partition_clause

The *add_range_partition_clause* lets you add a new range partition to the "high" end of a partitioned table (after the last existing partition). You can specify any create-time physical attributes for the new partition. If the table contains LOB columns, then you can also specify partition-level attributes for one or more LOB items.

If you do not specify a new *partition_name*, then Oracle assigns a name of the form SYS_P*n*. If you add a range partition to a composite-partitioned table and do not describe the subpartitions, then Oracle assigns subpartition names as described in *partition_level_subpartition* on page 11-69.

If a domain index is defined on *table*, then the index must not be marked IN_PROGRESS or FAILED.

A table can have up to 64K-1 partitions.

Restrictions on adding range partitions:

- If the upper partition bound of each partitioning key in the existing high partition is MAXVALUE, then you cannot add a partition to the table. Instead, use the *split_table_partition* clause to add a partition at the beginning or the middle of the table.
- The *key_compression* and OVERFLOW clauses are valid only for a partitioned index-organized table. You can specify OVERFLOW only if the partitioned table already has an overflow segment. You can specify key compression only if key compression is enabled at the table level.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.

range_values_clause Specify the upper bound for the new partition. The *value_list* is a comma-delimited, ordered list of literal values corresponding to *column_list*. The *value_list* must collate greater than the partition bound for the highest existing partition in the table.

partition_level_subpartition The *partition_level_subpartition* clause (in *table_partition_description*) is valid only for a composite-partitioned table. This clause lets you specify hash or list subpartitions for a new range-hash or range-list composite partition. This clause overrides any subpartition descriptions defined in *subpartition_template* at the table level.

For all composite partitions:

- You can specify the number of subpartitions (and optionally one or more tablespaces where they are to be stored). In this case, Oracle assigns subpartition names of the form SYS_SUBPn. The number of tablespaces does not have to equal the number of subpartitions. If the number of partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.
- Alternatively, you can use the *subpartition_spec* to specify individual subpartitions by name, and optionally the tablespace where each should be stored.
- If you omit *partition_level_subpartition* and if you have created a subpartition template, Oracle uses the template to create subpartitions. If you have not created a subpartition template, Oracle creates one hash subpartition or one DEFAULT list subpartition.

- If you omit *partition_level_subpartition* entirely, Oracle assigns subpartition names as follows:
  - If you have specified a subpartition template *and* you have specified partition names, then Oracle generates subpartition names of the form "*partition_name* underscore (_) *subpartition_name*" (for example, P1_SUB1).
  - If you have not specified a subpartition template *or* if you have specified a subpartition template but did not specify partition names, then Oracle generates subpartition names of the form SYS_SUBP*n*.
- In *partition_spec*, the only clause of the *partitioning_storage_clause* you can specify is the TABLESPACE clause.

For range-hash composite partitions, the *list_values_clause* of *subpartition_spec* is not relevant and is invalid.

For range-list composite partitions:

- The *hash_subpartition_quantity* is not relevant, so you must use the lower branch of *partition_level_subpartition*.
- Within *subpartition_spec*, you must specify the *list_values_clause* for each subpartition, and the values you specify for each subpartition cannot exist in any other subpartition of the same partition.

See Also: CREATE TABLE on page 15-7

Oracle will add a new index partition with the same subpartition descriptions to all local indexes defined on *table*. Global indexes defined on *table* are not affected.

add_hash_partition_clause

The *add_hash_partition_clause* lets you add a new hash partition to the "high" end of a partitioned table. Oracle will populate the new partition with rows rehashed from other partitions of *table* as determined by the hash function. For optimal load balancing, the total number of partitions should be a power of 2.

You can specify a name for the partition, and optionally a tablespace where it should be stored. If you do not specify a name, then Oracle assigns a partition name of the form SYS_P*n*. If you do not specify TABLESPACE, then the new partition is stored in the table's default tablespace. Other attributes are always inherited from table-level defaults.

You can update global indexes on *table* during this operation using the *update_global_index_clause*. For a heap-organized table, if this operation causes data to be rehashed among partitions, then Oracle marks UNUSABLE, and you must rebuild, any corresponding local index partitions. Indexes on index-organized tables are primary key based, so they do not become unusable.

Use the *parallel_clause* to specify whether to parallelize the creation of the new partition.

> **See Also:** CREATE TABLE on page 15-7 and *Oracle9i Database Concepts* for more information on hash partitioning

Restriction on adding hash partitions: In *table_partition_description*, you cannot specify *partition_level_subpartition*.

add_list_partition_clause

The *add_list_partition_clause* lets you add a new partition to *table* using a new set of partition values. You can specify any create-time physical attributes for the new partition. If the table contains LOB columns, then you can also specify partition-level attributes for one or more LOB items.

When you add a list partition to a table, Oracle adds a corresponding index partition with the same value list to all local indexes defined on the table. Global indexes are not affected.

Restrictions on adding list partitions:

- In *table_partition_description*, you cannot specify *partition_level_subpartition*.
- You cannot add a list partition if you have already defined a DEFAULT partition for the table. Instead you must use the *split_table_partition* clause to split the DEFAULT partition.

> **See Also:**
>
> - *list_partitioning* of CREATE TABLE on page 15-48 for more information and restrictions on list partitions
> - "Working with Default List Partitions: Example" on page 11-93

coalesce_table_partition

COALESCE applies only to hash partitions. Use the *coalesce_table_partition* clause to indicate that Oracle should select the last hash partition, distribute its

contents into one or more remaining partitions (determined by the hash function), and then drop the last partition.

Oracle invalidates any global indexes on heap-organized tables. You can update these indexes during this operation using the *update_global_index_clause*. Global indexes on index-organized tables are primary key based, so they do not become unusable.

Oracle drops local index partitions corresponding to the selected partition. Oracle marks UNUSABLE, and you must rebuild, the local index partitions corresponding to one or more absorbing partitions.

drop_table_partition

The *drop_table_partition* clause removes *partition*, and the data in that partition, from a partitioned table. If you want to drop a partition but keep its data in the table, then you must merge the partition into one of the adjacent partitions.

See Also: *merge_table_partitions* on page 11-78

If the table has LOB columns, then Oracle also drops the LOB data and LOB index partitions (and their subpartitions, if any) corresponding to *partition*.

If *table* is index organized and has a mapping table defined on it, then Oracle drops the corresponding mapping table partition as well.

Oracle drops local index partitions and subpartitions corresponding to *partition*, even if they are marked UNUSABLE.

You can update global indexes on heap-organized tables during this operation using the *update_global_index_clause*. If you specify the *parallel_clause* with the *update_global_index_clause*, then Oracle parallelizes the index update, not the drop operation.

If you drop a range partition and later insert a row that would have belonged to the dropped partition, then Oracle stores the row in the next higher partition. However, if that partition is the highest partition, then the insert will fail because the range of values represented by the dropped partition is no longer valid for the table.

Restrictions on dropping table partitions:

- You cannot drop a partition of a hash-partitioned table.
- If *table* contains only one partition, then you cannot drop the partition. You must drop the table.

See Also: "Dropping a Table Partition: Example" on page 11-94

drop_table_subpartition

Use this clause to drop a list subpartition from a range-list composite-partitioned table. Oracle deletes any rows in the dropped subpartition.

Oracle drops the corresponding subpartition of any local index. Other index subpartitions are not affected. Any global indexes are marked UNUSABLE unless you specify the *update_global_index_clause*.

Restrictions on dropping table subpartitions:

- You cannot drop a hash subpartition. Instead use the MODIFY PARTITION ... COALESCE SUBPARTITION syntax.
- You cannot drop the last subpartition of a partition. Instead use the *drop_table_partition* clause.

rename_partition_subpart

Use the *rename_table_partition* clause to rename a table partition or subpartition *current_name* to *new_name*. For both partitions and subpartitions, *new_name* must be different from all existing partitions and subpartitions of the same table.

If *table* is index organized, then Oracle assigns the same name to the corresponding primary key index partition as well as to any existing overflow partitions and mapping table partitions.

See Also: "Renaming Table Partitions: Examples" on page 11-94

truncate_partition_subpart

Specify TRUNCATE PARTITION to remove all rows from *partition* or, if the table is composite partitioned, all rows from *partition*'s subpartitions. Specify TRUNCATE SUBPARTITION to remove all rows from *subpartition*. If *table* is index organized, then Oracle also truncates any corresponding mapping table partitions and overflow area partitions.

If the partition or subpartition to be truncated contains data, then you must first disable any referential integrity constraints on the table. Alternatively, you can delete the rows and then truncate the partition.

If the table contains any LOB columns, then the LOB data and LOB index segments for this partition are also truncated. If *table* is composite partitioned, then the LOB data and LOB index segments for this partition's subpartitions are truncated.

If a domain index is defined on *table*, then the index must not be marked IN_PROGRESS or FAILED, and the index partition corresponding to the table partition being truncated must not be marked IN_PROGRESS.

For each partition or subpartition truncated, Oracle also truncates corresponding local index partitions and subpartitions. If those index partitions or subpartitions are marked UNUSABLE, then Oracle truncates them and resets the UNUSABLE marker to VALID.

You can update global indexes on *table* during this operation using the *update_global_index_clause*. If you specify the *parallel_clause* with the *update_global_index_clause*, then Oracle parallelizes the index update, not the truncate operation.

DROP STORAGE Specify DROP STORAGE to deallocate space from the deleted rows and make it available for use by other schema objects in the tablespace.

REUSE STORAGE Specify REUSE STORAGE to keep space from the deleted rows allocated to the partition or subpartition. The space is subsequently available only for inserts and updates to the same partition or subpartition.

See Also: "Truncating Table Partitions: Example" on page 11-95

split_table_partition

The *split_table_partition* clause lets you create, from *current_partition*, two new partitions, each with a new segment and new physical attributes, and new initial extents. The segment associated with *current_partition* is discarded.

The new partitions inherit all unspecified physical attributes from *current_partition*.

Note: Oracle can optimize and speed up SPLIT PARTITION and SPLIT SUBPARTITION operations if specific conditions are met. Please refer to *Oracle9i Database Administrator's Guide* for information on optimizing these operations.

If you split a DEFAULT list partition, then the first of the resulting partitions will have the split values, and the second resulting partition will have the DEFAULT value.

If *table* is index organized, then Oracle splits any corresponding mapping table partition and places it in the same tablespace as the parent index-organized table partition. Oracle also splits any corresponding overflow area, and you can specify segment attributes for the new overflow areas using the OVERFLOW clause.

Oracle splits the corresponding local index partition, even if it is marked UNUSABLE. Oracle marks UNUSABLE, and you must rebuild, the local index partitions corresponding to the split partitions. The new index partitions inherit their attributes from the partition being split. Oracle stores the new index partitions in the default tablespace of the index partition being split. If that index partition has no default tablespace, then Oracle uses the tablespace of the new underlying table partitions.

If *table* contains LOB columns, then you can use the *LOB_storage_clause* to specify separate LOB storage attributes for the LOB data segments resulting from the split. Oracle drops the LOB data and LOB index segments of *current_partition* and creates new segments for each LOB column, for each partition, even if you do not specify a new tablespace.

AT Clause The AT clause applies only to **range partitions**. Specify the new noninclusive upper bound for the first of the two new partitions. The value list must compare less than the original partition bound for *current_partition* and greater than the partition bound for the next lowest partition (if there is one).

VALUES Clause The VALUES clause applies only to **list partitions**. Specify the partition values you want to include in the first of the two new partitions. Oracle creates the first new partition using the partition value list you specify and creates the second new partition using the remaining partition values from *current_partition*. Therefore, the value list cannot contain all of the partition values of *current_partition*, nor can it contain any partition values that do not already exist for *current_partition*.

INTO Clause The INTO clause lets you describe the two partitions resulting from the split. In *function_spec*, the keyword PARTITION is required even if you do not specify the optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, then Oracle assigns names of the form SYS_P*n*. Any attributes you do not specify are inherited from *current_partition*.

For range-hash composite-partitioned tables, if you specify subpartitioning for the new partitions, then you can specify only TABLESPACE for the subpartitions. All other attributes are inherited from *current_partition*. If you do not specify subpartitioning for the new partitions, then their tablespace is also inherited from *current_partition*.

For range-list composite-partitioned tables, you cannot specify subpartitions for the new partitions at all (using the *partition_level_subpartition* clause of *table_partition_description*). The subpartitions of the split partition will inherit all their attributes (number of subpartitions and value lists) from c*urrent_partition*.

For all range-list composite-partitioned tables, and for range-hash composite-partitioned tables for which you do not specify subpartition names for the newly created subpartitions, the newly created subpartitions inherit their names from the parent partition as follows:

- For those subpartitions in the parent partition with names of the form "*partition_name* underscore (_) *subpartition_name*" (for example, P1_SUBP1), Oracle generates corresponding names in the newly created subpartitions using the new partition names (for example P1A_SUB1 and P1B_SUB1).
- For those subpartitions in the parent partition with names of any other form, Oracle generates subpartition names of the form SYS_SUBP*n*.

Oracle splits the corresponding partition in each local index defined on *table*, even if the index is marked UNUSABLE.

Oracle invalidates any global indexes on heap-organized tables. You can update these indexes during this operation using the *update_global_index_clause*. Global indexes on index-organized tables are primary key based, so they do not become unusable.

The *parallel_clause* lets you parallelize the split operation, but does not change the default parallel attributes of the table.

Restrictions on splitting table partitions

- You cannot specify this clause for a hash-partitioned table.
- In *partition_spec*, you can specify the *key_compression* clause and OVERFLOW clause only for a partitioned index-organized table. Also, you cannot specify the PCTUSED parameter for the index segment of an index-organized table.

See Also: "Splitting Table Partitions: Examples" on page 11-92 and "Working with Default List Partitions: Example" on page 11-93

split_table_subpartition

Use this clause to split a list subpartition into two separate subpartitions with nonoverlapping value lists.

Note: Oracle can optimize and speed up SPLIT PARTITION and SPLIT SUBPARTITION operations if specific conditions are met. Please refer to *Oracle9i Database Administrator's Guide* for information on optimizing these operations.

- In the VALUES clause, specify the subpartition values you want to include in the first of the two new subpartitions. You can specify NULL if you have not already specified NULL for another subpartition in the same partition. Oracle creates the first new subpartition using the subpartition value list you specify and creates the second new partition using the remaining partition values from the current subpartition. Therefore, the value list cannot contain all of the partition values of the current subpartition, nor can it contain any partition values that do not already exist for the current subpartition.
- The INTO clause lets you describe the two subpartitions resulting from the split. In *subpartition_spec*, the keyword PARTITION is required even if you do not specify the optional names and attributes of the two new subpartitions. If you do not specify new subpartition names, or if you omit this clause entirely, then Oracle assigns names of the form SYS_SUBP*n*. Any attributes you do not specify are inherited from the current subpartition.

Oracle splits any corresponding local subpartition index, even if it is marked UNUSABLE. The new index subpartitions will inherit the names of the new table subpartitions unless those names are already held by index subpartitions. In that case, Oracle assigns new index subpartition names of the form SYS_SUBPn. The new index subpartitions inherit physical attributes from the parent subpartition. However, if the parent subpartition does not have a default TABLESPACE attribute, then the new subpartitions inherit the tablespace of the corresponding new table subpartitions.

Oracle marks all global indexes on table UNUSABLE. If you also specify the *update_global_index_clause*, then Oracle will attempt to rebuild these global indexes.

Restrictions on splitting list subpartitions

- You cannot specify this clause for a hash subpartition.
- In *subpartition_spec*, the only clause of *partitioning_storage_clause* you can specify is the TABLESPACE clause.

merge_table_partitions

The *merge_table_partitions* clause lets you merge the contents of two partitions of *table* into one new partition, and then drops the original two partitions.

- The two partitions to be merged must be adjacent if they are range partitions. The new partition inherits the partition bound of the higher of the two original partitions.
- List partitions need not be adjacent in order to be merged. When you merge two list partitions, the resulting partition value list is the union of the set of the two partition values lists of the partitions being merged. If you merge a DEFAULT list partition with another list partition, then the resulting partition will be the DEFAULT partition and will have the DEFAULT value.
- When you merge two range-list composite partitions, you cannot specify the *partition_level_subpartition*. Oracle obtains the subpartitioning information from any subpartition template. If you have not specified a subpartition template, then Oracle creates exactly one DEFAULT subpartition.

Any attributes not specified in the *segment_attributes_clause* are inherited from table-level defaults.

If you do not specify a new *partition_name*, then Oracle assigns a name of the form SYS_P*n*. If the new partition has subpartitions, then Oracle assigns subpartition names as described in *partition_level_subpartition* on page 11-69.

Oracle marks UNUSABLE any global indexes on heap-organized tables. You can update these indexes during this operation using the *update_global_index_clause*. Global indexes on index-organized tables are primary key based, so they do not become unusable.

Oracle drops local index partitions corresponding to the selected partitions. Oracle marks UNUSABLE, and you must rebuild, the local index partition corresponding to merged partition.

Restriction on merging table partitions: You cannot specify this clause for a hash-partitioned table. Instead, use the *coalesce_table_partition* clause.

> **See Also:** "Merging Two Table Partitions: Example" on page 11-93 and "Working with Default List Partitions: Example" on page 11-93

partition_level_subpartition The *partition_level_subpartition* clause is valid only when you are merging range-hash composite partitions. This clause lets you specify subpartitioning attributes for the newly merged partition. Any attributes not specified in this clause are inherited from table-level values. If you do not specify this clause, then the new merged partition inherits subpartitioning attributes from table-level defaults.

If you omit this clause, then the new partition inherits the subpartition descriptions from any subpartition template you have defined. If you have not defined a subpartition template, then Oracle creates one subpartition in the newly merged partition.

Specify the *parallel_clause* to parallelize the merge operation.

Restriction on the *partition_level_subpartition* **clause:** You cannot specify this clause for a range-list composite partition.

merge_table_subpartitions

The *merge_table_subpartitions* clause lets you merge the contents of two list subpartitions of *table* into one new subpartition, and then drops the original two subpartitions. The two subpartitions to be merged must belong to the same partition, but they do not have to be adjacent. The data in the resulting subpartition will consist of the combined data from the merged subpartitions.

- If you do not specify a new subpartition name, or if you omit the INTO clause entirely, then Oracle assigns a name of the form SYS_SUBP*n*.
- If you do specify the INTO clause, then the keyword SUBPARTITION in *subpartition_spec* is required, you cannot specify the *list_values_clause*, and the only clause you can specify in the *partitioning_storage_clause* is the TABLESPACE clause.

Any attributes you do not specify explicitly for the new subpartition are inherited from partition-level values. If you reuse one of the subpartition names for the new subpartition, then the new subpartition will inherit values from the subpartition whose name is being reused rather than from partition-level default values.

Oracle merges corresponding local index subpartitions and marks the resulting index subpartition UNUSABLE. Oracle also marks UNUSABLE both partitioned and nonpartitioned global indexes on *table*.

Restriction on merging table subpartitions: You cannot specify this clause for a hash subpartition.

exchange_partition_subpart

Use the EXCHANGE PARTITION or EXCHANGE SUBPARTITION clause to exchange the data and index segments of:

- One nonpartitioned table with one hash, list, or range partition (or one hash or list subpartition)
- One hash-partitioned table with the hash subpartitions of a range partition of a range-hash composite-partitioned table
- One list-partitioned table with the list subpartitions of a range partition of a range-list composite-partitioned table.

In all cases, the structure of the table and the partition or subpartition being exchanged, including their partitioning keys, must be identical. In the case of list partitions and subpartitions, the corresponding value lists must also match.

This clause facilitates high-speed data loading when used with transportable tablespaces.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on transportable tablespaces

If *table* contains LOB columns, then for each LOB column Oracle exchanges LOB data and LOB index partition or subpartition segments with corresponding LOB data and LOB index segments of *table*.

All of the segment attributes of the two objects (including tablespace and logging) are also exchanged.

All statistics of the table and partition are exchanged, including table, column, index statistics, and histograms. The aggregate statistics of the table receiving the new partition are recalculated.

Oracle invalidates any global indexes on the objects being exchanged. If you specify the *update_global_index_clause* with this clause, then Oracle updates the global indexes on the table whose partition is being exchanged. Global indexes on the table being exchanged remain invalidated. If you specify the *parallel_*

clause with the *update_global_index_clause*, then Oracle parallelizes the index update, not the exchange operation.

See Also: "Restrictions on exchanging partitions:" on page 11-82

WITH TABLE *table* Specify the table with which the partition or subpartition will be exchanged.

INCLUDING INDEXES Specify INCLUDING INDEXES if you want local index partitions or subpartitions to be exchanged with the corresponding table index (for a nonpartitioned table) or local indexes (for a hash-partitioned table).

EXCLUDING INDEXES Specify EXCLUDING INDEXES if you want all index partitions or subpartitions corresponding to the partition and all the regular indexes and index partitions on the exchanged table to be marked UNUSABLE.

WITH VALIDATION Specify WITH VALIDATION if you want Oracle to return an error if any rows in the exchanged table do not map into partitions or subpartitions being exchanged.

WITHOUT VALIDATION Specify WITHOUT VALIDATION if you do not want Oracle to check the proper mapping of rows in the exchanged table.

exceptions_clause Specify a table into which Oracle places the rowids of all rows violating the constraint. If you omit *schema*, then Oracle assumes the exceptions table is in your own schema. If you omit this clause altogether, then Oracle assumes that the table is named EXCEPTIONS. The exceptions table must be on your local database.

You can create the EXCEPTIONS table using one of these scripts:

- UTLEXCPT.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLEXPT1.SQL uses universal rowids, so it can accommodate rows from both heap-organized and index-organized tables.

If you create your own exceptions table, then it must follow the format prescribed by one of these two scripts.

Note: If you are collecting exceptions from index-organized tables based on primary keys (rather than universal rowids), then you must create a separate exceptions table for each index-organized table to accommodate its primary key storage. You create multiple exceptions tables with different names by modifying and resubmitting the script.

See Also:

- The `DBMS_IOT` package in *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the SQL scripts
- *Oracle9i Database Performance Tuning Guide and Reference* for information on eliminating migrated and chained rows
- *Oracle9i Database Migration* for compatibility issues related to the use of these scripts

Restrictions on the *exceptions_clause:*

- This clause is not valid with subpartitions.
- The partitioned table must have been defined with a `UNIQUE` constraint, and that constraint must be in `DISABLE VALIDATE` state.

If these conditions are not true, then Oracle ignores this clause.

See Also: The *`constraints`* on page 7-5 for more information on constraint checking and "Creating an Exceptions Table for Index-Organized Tables: Example" on page 11-90

Restrictions on exchanging partitions:

- Both tables involved in the exchange must have the same primary key, and no validated foreign keys can be referencing either of the tables unless the referenced table is empty.
- When exchanging between a partitioned table and the range partition of a composite-partitioned table, the following restrictions apply:
  - The partitioning key of the partitioned table must be identical to the subpartitioning key of the composite-partitioned table.

  - The number of partitions in the partitioned table must be identical to the number of subpartitions in the range partition of the composite-partitioned table.
  - If you are exchanging a list-partitioned table with a range-list partition of a composite-partitioned table, then the values list of the list partitions must exactly match the values list of the range-list subpartitions.
- For partitioned index-organized tables, the following additional restrictions apply:
  - The source and target table / partition must have their primary key set on the same columns, in the same order.
  - If compression is enabled, then it must be enabled for both the source and the target, and with the same prefix length.
  - Both the source and target must be index organized.
  - Both the source and target must have overflow segments, or neither can have overflow segments. Also, both the source and target must have mapping tables, or neither can have a mapping table.
  - Both the source and target must have identical storage attributes for any LOB columns.

See Also: "Exchanging Table Partitions: Example" on page 11-94

UNUSABLE LOCAL INDEXES Clauses

These two clauses modify the attributes of local **index partitions** and **index subpartitions** corresponding to *partition* (depending on whether you are modifying a partition or subpartition).

- UNUSABLE LOCAL INDEXES marks UNUSABLE the local index partition or subpartition associated with *partition*.
- REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index partition or subpartition associated with *partition*.

Restrictions on the UNUSABLE LOCAL INDEXES clause:

- You cannot specify this clause with any other clauses of the *modify_table_partition* clause.
- You cannot specify this clause in the *modify_table_partition* clause for a partition that has subpartitions. However, you can specify this clause in the *modify_hash_subpartition* or *modify_list_subpartition* clause.

update_global_index_clause

When you perform DDL on a table partition, if a global index is defined on *table*, then Oracle invalidates the entire index, not just the partitions undergoing DDL. This clause lets you update the global index partition you are changing during the DDL operation, eliminating the need to rebuild the index after the DDL.

UPDATE GLOBAL INDEXES Specify UPDATE GLOBAL INDEXES to update the global indexes defined on *table*.

INVALIDATE GLOBAL INDEXES Specify INVALIDATE GLOBAL INDEXES to invalidate the global indexes defined on *table*.

If you specify neither, then Oracle invalidates the global indexes.

Restrictions on invalidating global indexes: This clause supports only global indexes. Domain indexes and index-organized tables are not supported. In addition, this clause updates only indexes that are USABLE and VALID. UNUSABLE indexes are left unusable, and INVALID global indexes are ignored.

> **See Also:** "Updating Global Indexes: Example" on page 11-95

parallel_clause

The *parallel_clause* lets you change the default degree of parallelism for queries and DML on the table.

> **Note:** The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL. Specify NOPARALLEL for serial execution. This is the default.

PARALLEL. Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer*. Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates

the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restrictions on the *parallel_clause*:

- If *table* contains any columns of LOB or user-defined object type, then subsequent INSERT, UPDATE, and DELETE operations on *table* are executed serially without notification. Subsequent queries, however, are executed in parallel.
- If you specify the *parallel_clause* in conjunction with the *move_table_clause*, then the parallelism applies only to the move, not to subsequent DML and query operations on the table.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56 and "Specifying Parallel Processing: Example" on page 11-88

move_table_clause

The *move_table_clause* lets you relocate data of a nonpartitioned table into a new segment, optionally in a different tablespace, and optionally modify any of its storage attributes.

You can also move any LOB data segments associated with the table using the *LOB_storage_clause* and *varray_col_properties* clause. LOB items not specified in this clause are not moved.

index_org_table_clause

For an index-organized table, the *index_org_table_clause* of the syntax lets you additionally specify overflow segment attributes. The *move_table_clause* rebuilds the index-organized table's primary key index. The overflow data segment is not rebuilt unless the OVERFLOW keyword is explicitly stated, with two exceptions:

- If you alter the values of PCTTHRESHOLD or the INCLUDING column as part of this ALTER TABLE statement, then the overflow data segment is rebuilt.
- If you explicitly move any of out-of-line columns (LOBs, varrays, nested table columns) in the index-organized table, then the overflow data segment is also rebuilt.

The index and data segments of LOB columns are not rebuilt unless you specify the LOB columns explicitly as part of this ALTER TABLE statement.

ONLINE Clause Specify ONLINE if you want DML operations on the index-organized table to be allowed during rebuilding of the table's primary key index.

Restrictions on the ONLINE clause:

- You cannot combine this clause with any other clause in the same statement.
- You can specify this clause only for a nonpartitioned index-organized table.
- Parallel DML is not supported during online MOVE. If you specify ONLINE and then issue parallel DML statements, then Oracle returns an error.

mapping_table_clauses Specify MAPPING TABLE if you want Oracle to create a mapping table if one does not already exist. If it does exist, then Oracle moves the mapping table along with the index-organized table, and marks any bitmapped indexes UNUSABLE. The new mapping table is created in the same tablespace as the parent table.

Specify NOMAPPING to instruct Oracle to drop an existing mapping table.

Restriction on mapping tables: You cannot specify NOMAPPING if any bitmapped indexes have been defined on table.

> **See Also:** *mapping_table_clause* of CREATE TABLE on page 15-32

key_compression Use the *key_compression* clause to enable or disable key compression in an index-organized table.

- COMPRESS enables key compression, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length (number of prefix columns to compress).

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.

- NOCOMPRESS disables key compression in index-organized tables. This is the default.

TABLESPACE *tablespace* Specify the tablespace into which the rebuilt index-organized table is stored.

Restrictions on the *move_table_clause*:

- If you specify MOVE, then it must be the first clause, and the only clauses outside this clause that are allowed are the *physical_attributes_clause*, the *parallel_clause*, and the *LOB_storage_clause*.
- You cannot move a table containing a LONG or LONG RAW column.
- You cannot MOVE an entire partitioned table (either heap or index organized). You must move individual partitions or subpartitions.

> **See Also:** *move_table_partition* on page 11-66 and *move_table_subpartition* on page 11-67

Notes regarding LOBs:

For any LOB columns you specify in a *move_table_clause*:

- Oracle drops the old LOB data segment and corresponding index segment and creates new segments, even if you do not specify a new tablespace.
- If the LOB index in *table* resided in a different tablespace from the LOB data, then Oracle collocates the LOB index with the LOB data in the LOB data's tablespace after the move.

---

enable_disable_clause

The *enable_disable_clause* lets you specify whether and how Oracle should apply an integrity constraint. The DROP and KEEP clauses are valid only when you are disabling a unique or primary key constraint.

> **See Also:**
>
> - The *enable_disable_clause* of CREATE TABLE on page 15-57 for a complete description of this clause, including notes and restrictions that relate to this statement
> - "Using Indexes to Enforce Constraints" on page 7-22 for information on using indexes to enforce constraints

TABLE LOCK

Oracle permits DDL operations on a table only if the table can be locked during the operation. Such table locks are not required during DML operations.

Note: Table locks are not acquired on temporary tables.

ENABLE TABLE LOCK Specify ENABLE TABLE LOCK to enable table locks, thereby allowing DDL operations on the table.

DISABLE TABLE LOCK Specify DISABLE TABLE LOCK to disable table locks, thereby preventing DDL operations on the table.

ALL TRIGGERS

Use the ALL TRIGGERS clause to enable or disable all triggers associated with the table.

ENABLE ALL TRIGGERS Specify ENABLE ALL TRIGGERS to enable all triggers associated with the table. Oracle fires the triggers whenever their triggering condition is satisfied.

To enable a single trigger, use the *enable_clause* of ALTER TRIGGER.

> **See Also:** CREATE TRIGGER on page 15-95, ALTER TRIGGER on page 12-2, and "Enabling Triggers: Example" on page 11-90

DISABLE ALL TRIGGERS Specify DISABLE ALL TRIGGERS to disable all triggers associated with the table. Oracle will not fire a disabled trigger even if the triggering condition is satisfied.

Examples

Collection Retrieval: Example The following statement modifies nested table column ad_textdocs_ntab in the sample table sh.print_media so that when queried it returns actual values instead of locators:

```
ALTER TABLE print_media MODIFY NESTED TABLE ad_textdocs_ntab
   RETURN AS VALUE;
```

Specifying Parallel Processing: Example The following statement specifies parallel processing for queries to the sample table oe.customers:

```
ALTER TABLE customers
   PARALLEL;
```

Changing the State of a Constraint: Examples The following statement places in ENABLE VALIDATE state an integrity constraint named emp_manager_fk in the employees table:

```
ALTER TABLE employees
   ENABLE VALIDATE CONSTRAINT emp_manager_fk
   EXCEPTIONS INTO exceptions;
```

Each row of the employees table must satisfy the constraint for Oracle to enable the constraint. If any row violates the constraint, then the constraint remains disabled. Oracle lists any exceptions in the table exceptions. You can also identify the exceptions in the employees table with the following statement:

```
SELECT employees.*
   FROM employees e, exceptions ex
   WHERE e.row_id = ex.row_id
      AND ex.table_name = 'EMPLOYEES'
      AND ex.constraint = 'EMP_MANAGER_FK';
```

The following statement tries to place in ENABLE NOVALIDATE state two constraints on the employees table:

```
ALTER TABLE employees
   ENABLE NOVALIDATE PRIMARY KEY
   ENABLE NOVALIDATE CONSTRAINT emp_last_name_nn;
```

This statement has two ENABLE clauses:

- The first places a primary key constraint on the table in ENABLE NOVALIDATE state.
- The second places the constraint named emp_last_name_nn in ENABLE NOVALIDATE state.

In this case, Oracle enables the constraints only if both are satisfied by each row in the table. If any row violates either constraint, then Oracle returns an error and both constraints remain disabled.

Consider a referential integrity constraint involving a foreign key on the combination of the areaco and phoneno columns of the phone_calls table. The foreign key references a unique key on the combination of the areaco and phoneno columns of the customers table. The following statement disables the unique key on the combination of the areaco and phoneno columns of the customers table:

```
ALTER TABLE customers
```

```
   DISABLE UNIQUE (areaco, phoneno) CASCADE;
```

The unique key in the `customers` table is referenced by the foreign key in the `phone_calls` table, so you must use the `CASCADE` clause to disable the unique key. This clause disables the foreign key as well.

Creating an Exceptions Table for Index-Organized Tables: Example The following example creates the `except_table` table to hold rows from the index-organized table `hr.countries` that violate the primary key constraint:

```
EXECUTE DBMS_IOT.BUILD_EXCEPTIONS_TABLE ('hr', 'countries', 'except_table');

ALTER TABLE countries
   ENABLE PRIMARY KEY
   EXCEPTIONS INTO except_table;
```

To specify an exception table, you must have the privileges necessary to insert rows into the table. To examine the identified exceptions, you must have the privileges necessary to query the exceptions table.

See Also:

- INSERT on page 17-54
- SELECT on page 18-4 for information on the privileges necessary to insert rows into tables

Disabling a CHECK Constraint: Example The following statement defines and disables a `CHECK` constraint on the `employees` table:

```
ALTER TABLE employees ADD CONSTRAINT check_comp
   CHECK (salary + (commission_pct*salary) <= 5000)
   DISABLE CONSTRAINT check_comp;
```

The constraint `check_comp` ensures that no employee's total compensation exceeds $5000. The constraint is disabled, so you can increase an employee's compensation above this limit.

Enabling Triggers: Example The following statement enables all triggers associated with the `employees` table:

```
ALTER TABLE employees
   ENABLE ALL TRIGGERS;
```

Deallocating Unused Space: Example The following statement frees all unused space for reuse in table employees, where the high water mark is above MINEXTENTS:

```
ALTER TABLE employees
    DEALLOCATE UNUSED;
```

Renaming a Column: Example The following example renames the credit_limit column of the sample table oe.customers to credit_amount:

```
ALTER TABLE customers
   RENAME COLUMN credit_limit TO credit_amount;
```

Dropping a Column: Example This statement illustrates the *drop_column_clause* with CASCADE CONSTRAINTS. Assume table t1 is created as follows:

```
CREATE TABLE t1 (
   pk NUMBER PRIMARY KEY,
   fk NUMBER,
   c1 NUMBER,
   c2 NUMBER,
   CONSTRAINT ri FOREIGN KEY (fk) REFERENCES t1,
   CONSTRAINT ck1 CHECK (pk > 0 and c1 > 0),
   CONSTRAINT ck2 CHECK (c2 > 0)
);
```

An error will be returned for the following statements:

```
/* The next two statements return errors:
ALTER TABLE t1 DROP (pk); -- pk is a parent key
ALTER TABLE t1 DROP (c1);  -- c1 is referenced by multicolumn
                           -- constraint ck1
```

Submitting the following statement drops column pk, the primary key constraint, the foreign key constraint, ri, and the check constraint, ck1:

```
ALTER TABLE t1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to column pk, then it is valid to submit the following statement without the CASCADE CONSTRAINTS clause:

```
ALTER TABLE t1 DROP (pk, fk, c1);
```

Modifying Index-Organized Tables: Examples This statement modifies the INITRANS parameter for the index segment of index-organized table hr.countries:

```
ALTER TABLE countries INITRANS 4;
```

The following statement adds an overflow data segment to index-organized table countries:

```
ALTER TABLE countries ADD OVERFLOW;
```

This statement modifies the INITRANS parameter for the overflow data segment of index-organized table countries:

```
ALTER TABLE countries OVERFLOW INITRANS 4;
```

Splitting Table Partitions: Examples The following statement splits the old partition sales_q4_2000 in the sample table sh.sales, creating two new partitions, naming one sales_q4_2000b and reusing the name of the old partition for the other:

```
ALTER TABLE sales SPLIT PARTITION SALES_Q4_2000
   AT (TO_DATE('15-NOV-2000','DD-MON-YYYY'))
   INTO (PARTITION SALES_Q4_2000, PARTITION SALES_Q4_2000b);
```

Assume that the sample table pm.print_media was range partitioned into partitions p1 and p2. (You would have to convert the LONG column in print_media to LOB before partitioning the table.) The following statement splits partition p2 of that table into partitions p2a and p2b:

```
ALTER TABLE print_media_part
   SPLIT PARTITION p2 AT (150) INTO
   (PARTITION p2a TABLESPACE omf_ts1
      LOB ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2),
   PARTITION p2b
      LOB (ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2));
```

In both partitions p2a and p2b, Oracle creates the LOB segments for columns ad_photo and ad_composite in tablespace omb_ts2. The LOB segments for the remaining columns in partition p2a are stored in tablespace omf_ts1. The LOB segments for the remaining columns in partition p2b remain in the tablespaces in which they resided prior to this ALTER statement. However, Oracle creates new segments for all the LOB data and LOB index segments, even if they are not moved to a new tablespace.

Adding a Table Partition with a LOB: Examples The following statement adds a partition p3 to the print_media_part table (see preceding example) and specifies storage characteristics for the table's BLOB and CLOB columns:

```
ALTER TABLE print_media_part ADD PARTITION p3 VALUES LESS THAN
(MAXVALUE)
   LOB (ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts2)
   LOB (ad_sourcetext, ad_finaltext) STORE AS (TABLESPACE omf_ts1);
```

The LOB data and LOB index segments for columns ad_photo and ad_composite in partition p3 will reside in tablespace omf_ts2. The remaining attributes for these LOB columns will be inherited first from the table-level defaults, and then from the tablespace defaults.

The LOB data segments for columns ad_source_text and ad_finaltext will reside in the omf_ts1 tablespace, and will inherit all other attributes from the table-level defaults and then from the tablespace defaults.

Working with Default List Partitions: Example The following statements use the list partitioned table created in "List Partitioning Example" on page 15-73. The first statement splits the existing default partition into a new south partition and a default partition:

```
ALTER TABLE list_customers SPLIT PARTITION rest
   VALUES ('MEXICO', 'COLOMBIA')
   INTO (PARTITION south, PARTITION rest);
```

The next statement merges the resulting default partition with the asia partition:

```
ALTER TABLE list_customers
   MERGE PARTITIONS asia, rest INTO PARTITION rest;
```

The next statement re-creates the asia partition by splitting the default partition:

```
ALTER TABLE list_customers SPLIT PARTITION rest
   VALUES ('CHINA', 'THAILAND')
   INTO (PARTITION east, partition rest);
```

Merging Two Table Partitions: Example The following statement merges back into one partition the partitions created in "Splitting Table Partitions: Examples" on page 11-92:

```
ALTER TABLE sales
   MERGE PARTITIONS sales_q4_2000, sales_q4_2000b
   INTO PARTITION sales_q4_2000;
```

Dropping a Table Partition: Example The following statement drops partition p3 created in "Adding a Table Partition with a LOB: Examples" on page 11-93:

```
ALTER TABLE print_media_part DROP PARTITION p3;
```

Exchanging Table Partitions: Example The following statement converts partition feb97 to table sales_feb97 without exchanging local index partitions with corresponding indexes on sales_feb97 and without verifying that data in sales_feb97 falls within the bounds of partition feb97:

```
ALTER TABLE sales
   EXCHANGE PARTITION feb97 WITH TABLE sales_feb97
   WITHOUT VALIDATION;
```

Modifying Table Partitions: Examples The following statement marks all the local index partitions corresponding to the nov96 partition of the sales table UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION nov96
   UNUSABLE LOCAL INDEXES;
```

The following statement rebuilds all the local index partitions that were marked UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION jan97
   REBUILD UNUSABLE LOCAL INDEXES;
```

The following statement changes MAXEXTENTS and logging attribute for partition branch_ny:

```
ALTER TABLE branch MODIFY PARTITION branch_ny
   STORAGE (MAXEXTENTS 75) LOGGING;
```

Moving Table Partitions: Example The following statement moves partition p2b (from "Splitting Table Partitions: Examples" on page 11-92) to tablespace omf_ts1:

```
ALTER TABLE print_media_part
   MOVE PARTITION p2b TABLESPACE omf_ts1;
```

Renaming Table Partitions: Examples The following statement renames a table:

```
ALTER TABLE employees RENAME TO employee;
```

In the following statement, partition emp3 is renamed:

```
ALTER TABLE employee RENAME PARTITION emp3 TO employee3;
```

Truncating Table Partitions: Example The following statement deletes all the data in the sys_p017 partition and deallocates the freed space:

```
ALTER TABLE deliveries
   TRUNCATE PARTITION sys_p017 DROP STORAGE;
```

Updating Global Indexes: Example The following statement splits partition sales_q1_2000 of the sample table sh.sales, and updates any global indexes defined on it:

```
ALTER TABLE sales SPLIT PARTITION sales_q1_2000
   AT (TO_DATE('16-FEB-2000','DD-MON-YYYY'))
   INTO (PARTITION q1a_2000, PARTITION q1b_2000)
   UPDATE GLOBAL INDEXES;
```

Specifying Object Identifiers: Example The following statements create an object type, a corresponding object table with a primary-key-based object identifier, and a table having a user-defined REF column:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));

CREATE TABLE emp OF emp_t (
   empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;

CREATE TABLE dept (dno NUMBER, mgr_ref REF emp_t SCOPE is emp);
```

The next statements add a constraint and a user-defined REF column, both of which reference table emp:

```
ALTER TABLE dept ADD CONSTRAINT mgr_cons FOREIGN KEY (mgr_ref)
   REFERENCES emp;
ALTER TABLE dept ADD sr_mgr REF emp_t REFERENCES emp;
```

Adding a Table Column: Example The following statement adds a column named duty_pct of datatype NUMBER and a column named visa_needed of datatype VARCHAR2 with a size of 3 (to hold "yes" and "no" data) and a CHECK integrity constraint:

```
ALTER TABLE countries
   ADD (duty_pct     NUMBER(2,2)  CHECK (duty_pct < 10.5),
        visa_needed  VARCHAR2(3));
```

Modifying Table Columns: Examples The following statement increases the size of the duty_pct column:

```
ALTER TABLE countries
   MODIFY (duty_pct NUMBER(3,2));
```

Because the MODIFY clause contains only one column definition, the parentheses around the definition are optional.

The following statement changes the values of the PCTFREE and PCTUSED parameters for the employees table to 30 and 60, respectively:

```
ALTER TABLE employees
   PCTFREE 30
   PCTUSED 60;
```

Converting LONG Columns to LOB: Example The following example modifies the press_release column of the sample table pm.print_media from LONG to CLOB datatype:

```
ALTER TABLE print_media MODIFY (press_release CLOB);
```

Allocating Extents: Example The following statement allocates an extent of 5 kilobytes for the employees table and makes it available to instance 4:

```
ALTER TABLE employees
  ALLOCATE EXTENT (SIZE 5K INSTANCE 4);
```

Because this statement omits the DATAFILE parameter, Oracle allocates the extent in one of the datafiles belonging to the tablespace containing the table.

Specifying Default Column Value: Examples This statement modifies the min_price column of the product_information table so that it has a default value of 10:

```
ALTER TABLE product_information
  MODIFY (min_price DEFAULT 10);
```

If you subsequently add a new row to the product_information table and do not specify a value for the min_price column, then the value of the min_price column is automatically 0:

```
INSERT INTO product_information (product_id, product_name,
   list_price)
   VALUES (300, 'left-handed mouse', 40.50);

SELECT product_id, product_name, list_price, min_price
    FROM product_information
    WHERE product_id = 300;
```

```
PRODUCT_ID PRODUCT_NAME         LIST_PRICE  MIN_PRICE
---------- -------------------- ---------- ----------
       300 left-handed mouse          40.5         10
```

To discontinue previously specified default values, so that they are no longer automatically inserted into newly added rows, replace the values with nulls, as shown in this statement:

```
ALTER TABLE product_information
   MODIFY (min_price DEFAULT NULL);
```

The MODIFY clause need only specify the column name and the modified part of the definition, rather than the entire column definition. This statement has no effect on any existing values in existing rows.

Adding a Constraint to an XMLType Table: Example The following example adds a primary key constraint to the xwarehouses table, created in "XMLType Table Examples" on page 15-71:

```
ALTER TABLE xwarehouses
   ADD (PRIMARY KEY(XMLDATA."WarehouseID"));
```

> **See Also:** XMLDATA on page 2-90 for information about this pseudocolumn

Renaming Constraints: Example The following statement renames the cust_fname_nn constraint on the sample table oe.customers to cust_firstname_nn:

```
ALTER TABLE customers RENAME CONSTRAINT cust_fname_nn
   TO cust_firstname_nn;
```

Dropping Constraints: Examples The following statement drops the primary key of the departments table:

```
ALTER TABLE departments
    DROP PRIMARY KEY CASCADE;
```

If you know that the name of the PRIMARY KEY constraint is pk_dept, then you could also drop it with the following statement:

```
ALTER TABLE departments
    DROP CONSTRAINT pk_dept CASCADE;
```

The CASCADE clause drops any foreign keys that reference the primary key.

The following statement drops the unique key on the email column of the employees table:

```
ALTER TABLE employees
    DROP UNIQUE (email);
```

The DROP clause in this statement omits the CASCADE clause. Because of this omission, Oracle does not drop the unique key if any foreign key references it.

LOB Columns: Examples The following statement adds CLOB column resume to the employee table and specifies LOB storage characteristics for the new column:

```
ALTER TABLE employees ADD (resume CLOB)
  LOB (resume) STORE AS resume_seg (TABLESPACE demo);
```

To modify the LOB column resume to use caching, enter the following statement:

```
ALTER TABLE employees MODIFY LOB (resume) (CACHE);
```

Nested Tables: Examples The following statement adds the nested table column skills to the employee table:

```
ALTER TABLE employees ADD (skills skill_table_type)
    NESTED TABLE skills STORE AS nested_skill_table;
```

You can also modify a nested table's storage characteristics. Use the name of the storage table specified in the *nested_table_col_properties* to make the modification. You cannot query or perform DML statements on the storage table. Use the storage table only to modify the nested table column storage characteristics.

The following statement creates table vetservice with nested table column client and storage table client_tab. Nested table vetservice is modified to specify constraints:

```
CREATE TYPE pet_table AS OBJECT
   (pet_name VARCHAR2(10), pet_dob DATE);

CREATE TABLE vetservice (vet_name VARCHAR2(30),
                         client   pet_table)
  NESTED TABLE client STORE AS client_tab;

ALTER TABLE client_tab ADD UNIQUE (ssn);
```

The following statement adds a UNIQUE constraint to nested table nested_skill_table:

```
ALTER TABLE nested_skill_table ADD UNIQUE (a);
```

The following statement alters the storage table for a nested table of REF values to specify that the REF is scoped:

```
CREATE TYPE emp_t AS OBJECT (eno number, ename char(31));
CREATE TYPE emps_t AS TABLE OF REF emp_t;
CREATE TABLE emptab OF emp_t;
CREATE TABLE dept (dno NUMBER, employees emps_t)
   NESTED TABLE employees STORE AS deptemps;
ALTER TABLE deptemps ADD (SCOPE FOR (column_value) IS emptab);
```

Similarly, to specify storing the REF with rowid:

```
ALTER TABLE deptemps ADD (REF(column_value) WITH ROWID);
```

In order to execute these ALTER TABLE statements successfully, the storage table deptemps must be empty. Also, because the nested table is defined as a table of scalars (REFs), Oracle implicitly provides the column name COLUMN_VALUE for the storage table.

See Also:

- CREATE TABLE on page 15-7 for more information about nested table storage
- *Oracle9i Application Developer's Guide - Fundamentals* for more information about nested tables

REF Columns: Examples In the following statement an object type dept_t has been previously defined. Now, create table staff as follows:

```
CREATE TABLE staff
   (name   VARCHAR(100),
    salary NUMBER,
    dept   REF dept_t);
```

An object table offices is created as:

```
CREATE TABLE offices OF dept_t;
```

The dept column can store references to objects of dept_t stored in any table. If you would like to restrict the references to point only to objects stored in the

departments table, then you could do so by adding a scope constraint on the dept column as follows:

```
ALTER TABLE staff
    ADD (SCOPE FOR (dept) IS offices);
```

The preceding ALTER TABLE statement will succeed only if the staff table is empty.

If you want the REF values in the dept column of staff to also store the rowids, issue the following statement:

```
ALTER TABLE staff
   ADD (REF(dept) WITH ROWID);
```

Additional Examples For examples of defining integrity constraints with the ALTER TABLE statement, see the *constraints* on page 7-5.

For examples of changing the value of a table's storage parameters, see the *storage_clause* on page 7-56.

ALTER TABLESPACE

Purpose

Use the ALTER TABLESPACE statement to alter an existing tablespace or one or more of its datafiles or tempfiles.

> **See Also:** *Oracle9i Database Administrator's Guide* and CREATE TABLESPACE on page 15-80 for information on creating a tablespace

Prerequisites

If you have ALTER TABLESPACE system privilege, then you can perform any of this statement's operations. If you have MANAGE TABLESPACE system privilege, then you can only perform the following operations:

- Take the tablespace online or offline
- Begin or end a backup
- Make the tablespace read only or read write

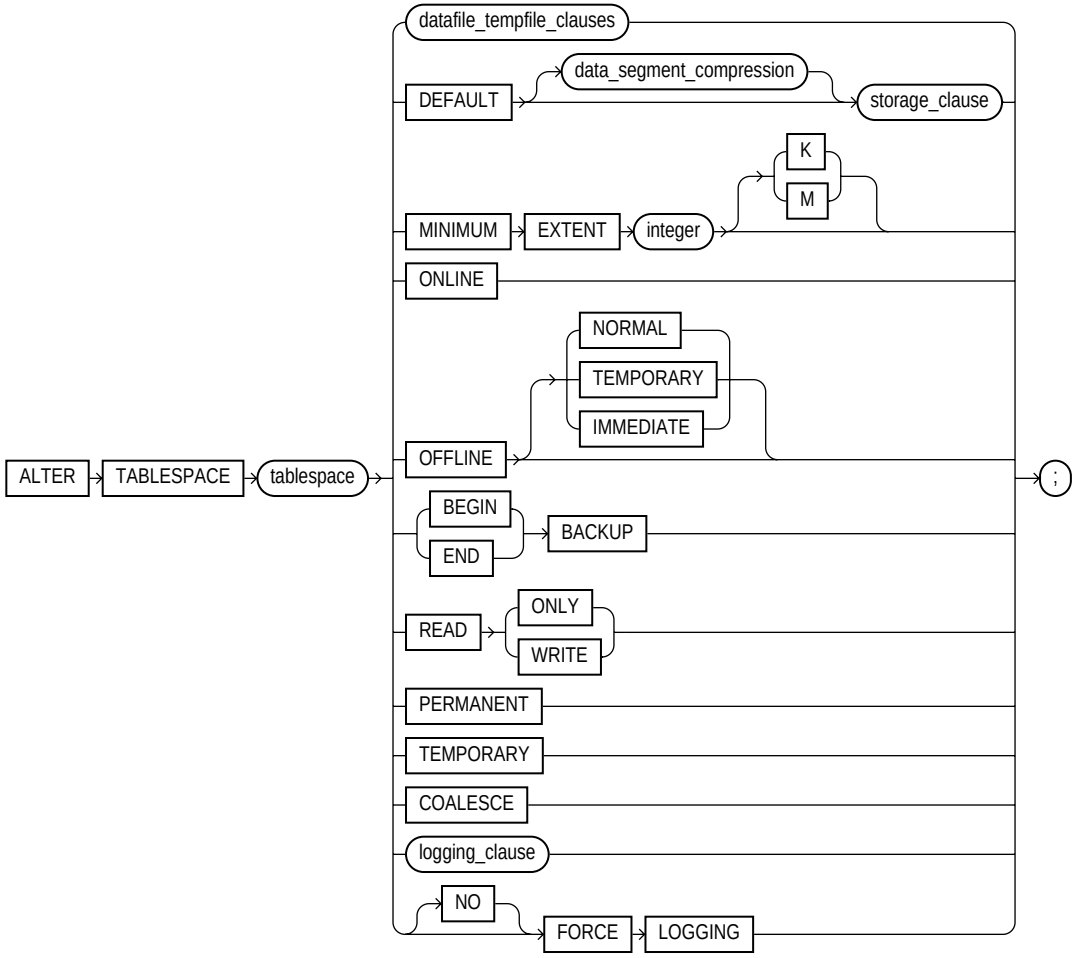
Before you can make a tablespace read only, the following conditions must be met:

- The tablespace must be online.
- The tablespace must not contain any active rollback segments. For this reason, the SYSTEM tablespace can never be made read only, because it contains the SYSTEM rollback segment. Additionally, because the rollback segments of a read-only tablespace are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace read only.
- The tablespace must not be involved in an open backup, because the end of a backup updates the header file of all datafiles in the tablespace.

Performing this function in restricted mode may help you meet these restrictions, because only users with RESTRICTED SESSION system privilege can be logged on.

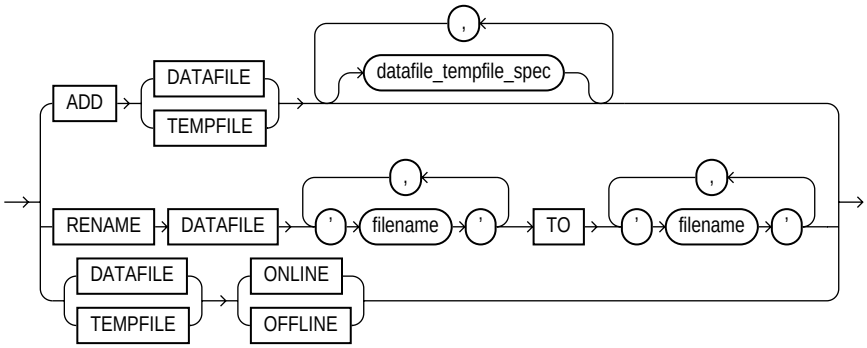
Syntax

alter_tablespace::=



(*datafile_tempfile_clauses::=* on page 11-103, *data_segment_compression::=* on page 11-5—part of ALTER TABLE syntax, *storage_clause::=* on page 7-58, *logging_clause::=* on page 7-46)

datafile_tempfile_clauses::=



(*datafile_tempfile_spec::=* on page 7-39—part of *file_specification*).

Keywords and Parameters

tablespace

Specify the name of the tablespace to be altered.

Restrictions on tablespaces:

- If *tablespace* is an undo tablespace, then the only other clauses you can specify in this statement are ADD DATAFILE, RENAME DATAFILE, DATAFILE ... ONLINE | OFFLINE, and BEGIN | END BACKUP.
- For locally managed temporary tablespaces the only clause you can specify in this statement is the ADD clause.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on Automatic Undo Management and undo tablespaces

datafile_tempfile_clauses

The tablespace file clauses let you add or modify a datafile or tempfile.

ADD DATAFILE | TEMPFILE Clause

Specify ADD to add to the tablespace a datafile or tempfile specified by *datafile_tempfile_spec*.

For locally managed temporary tablespaces, this is the only clause you can specify at any time.

If you omit *datafile_tempfile_spec*, then Oracle creates an Oracle-managed file of 100M with AUTOEXTEND enabled.

You can add a datafile or tempfile to a locally managed tablespace that is online or to a dictionary managed tablespace that is online or offline. Be sure the file is not in use by another database.

Note: On some operating systems, Oracle does not allocate space for the tempfile until the tempfile blocks are actually accessed. This delay in space allocation results in faster creation and resizing of tempfiles, but it requires that sufficient disk space is available when the tempfiles are later used. Please refer to the Oracle documentation for your operating system to determine whether Oracle allocates tempfile space in this way on your system.

See Also: *file_specification* on page 7-39, "Adding a Datafile: Example" on page 11-109, and "Adding an Oracle-managed Datafile: Example" on page 11-110

RENAME DATAFILE Clause

Specify RENAME DATAFILE to rename one or more of the tablespace's datafiles. The database must be open, and you must take the tablespace offline before renaming it. Each '*filename*' must fully specify a datafile using the conventions for filenames on your operating system.

This clause merely associates the tablespace with the new file rather than the old one. This clause does not actually change the name of the operating system file. You must change the name of the file through your operating system.

See Also: "Moving and Renaming Tablespaces: Example" on page 11-109

DATAFILE | TEMPFILE ONLINE | OFFLINE

Use this clause to take all datafiles or tempfiles in the tablespace offline or put them online. This clause has no effect on the ONLINE/OFFLINE status of the tablespace.

The database must be mounted. If tablespace is SYSTEM, or an undo tablespace, or the default temporary tablespace, then the database must not be open.

DEFAULT *storage_clause*

DEFAULT *storage_clause* lets you specify the new default storage parameters for objects subsequently created in the tablespace. For a dictionary-managed temporary table, Oracle considers only the NEXT parameter of the *storage_clause*.

Restriction on default storage: You cannot specify this clause for a locally managed tablespace.

> **See Also:** *storage_clause* on page 7-56

MINIMUM EXTENT

The MINIMUM EXTENT clause lets you control free space fragmentation in the tablespace by ensuring that every used or free extent in a tablespace is at least as large as, and is a multiple of, *integer*. This clause is not relevant for a dictionary-managed temporary tablespace.

Restriction on MINIMUM EXTENT: You cannot specify this clause for a locally managed tablespace.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information about using MINIMUM EXTENT to control space fragmentation and "Changing Tablespace Extent Allocation: Example" on page 11-110

ONLINE

Specify ONLINE to bring the tablespace online.

OFFLINE

Specify OFFLINE to take the tablespace offline and prevent further access to its segments. When you take a tablespace offline, all of its datafiles are also offline.

Suggestion: Before taking a tablespace offline for a long time, you may want to alter the tablespace allocation of any users who have been assigned the tablespace as either a default or temporary tablespace. When the tablespace is offline, these users cannot allocate space for objects or sort areas in the tablespace. See ALTER USER on page 12-21 for more information on allocating tablespace quota to users.

Restriction on the OFFLINE clause: You cannot take a temporary tablespace offline.

NORMAL Specify NORMAL to flush all blocks in all datafiles in the tablespace out of the SGA. You need not perform media recovery on this tablespace before bringing it back online. This is the default.

TEMPORARY If you specify TEMPORARY, then Oracle performs a checkpoint for all online datafiles in the tablespace but does not ensure that all files can be written. Any offline files may require media recovery before you bring the tablespace back online.

IMMEDIATE If you specify IMMEDIATE, then Oracle does not ensure that tablespace files are available and does not perform a checkpoint. You must perform media recovery on the tablespace before bringing it back online.

Note: The FOR RECOVER setting for ALTER TABLESPACE ... OFFLINE has been deprecated. The syntax is supported for backward compatibility. However, users are encouraged to use the transportable tablespaces feature for tablespace recovery.

See Also: *Oracle9i User-Managed Backup and Recovery Guide* for information on using transportable tablespaces to perform media recovery

BEGIN BACKUP

Specify BEGIN BACKUP to indicate that an open backup is to be performed on the datafiles that make up this tablespace. This clause does not prevent users from accessing the tablespace. You must use this clause before beginning an open backup.

Restrictions on the BEGIN BACKUP clause: You cannot specify this clause for a read-only tablespace or for a temporary locally managed tablespace.

Note: While the backup is in progress, you cannot take the tablespace offline normally, shut down the instance, or begin another backup of the tablespace.

See Also: "Backing Up Tablespaces: Examples" on page 11-109

END BACKUP

Specify END BACKUP to indicate that an online backup of the tablespace is complete. Use this clause as soon as possible after completing an online backup. Otherwise, if an instance failure or SHUTDOWN ABORT occurs, then Oracle assumes that media recovery (possibly requiring archived redo log) is necessary at the next instance start up.

Restriction on END BACKUP: You cannot use this clause on a read-only tablespace.

See Also:

- *Oracle9i Database Administrator's Guide* for information on restarting the database without media recovery
- ALTER DATABASE "END BACKUP Clause" on page 9-35 for information on taking individual datafiles (or all datafiles in the tablespace) out of online ("hot") backup mode

READ ONLY | READ WRITE

Specify READ ONLY to place the tablespace in **transition read-only mode**. In this state, existing transactions can complete (commit or roll back), but no further write operations (DML) are allowed to the tablespace except for rollback of existing transactions that previously modified blocks in the tablespace.

Once a tablespace is read only, you can copy its files to read-only media. You must then rename the datafiles in the control file to point to the new location by using the SQL statement ALTER DATABASE ... RENAME.

See Also:

- *Oracle9i Database Concepts* for more information on read-only tablespaces
- ALTER DATABASE on page 9-13

Specify READ WRITE to indicate that write operations are allowed on a previously read-only tablespace.

PERMANENT | TEMPORARY

Specify PERMANENT to indicate that the tablespace is to be converted from a temporary to a permanent one. A permanent tablespace is one in which permanent database objects can be stored. This is the default when a tablespace is created.

Specify TEMPORARY to indicate specifies that the tablespace is to be converted from a permanent to a temporary one. A temporary tablespace is one in which no permanent database objects can be stored. Objects in a temporary tablespace persist only for the duration of the session.

Restrictions on TEMPORARY:

- If tablespace was not created with a standard block size, then you cannot change it from permanent to temporary.
- You cannot specify TEMPORARY for a tablespace in FORCE LOGGING mode.

COALESCE

For each datafile in the tablespace, this clause combines all contiguous free extents into larger contiguous extents.

logging_clause

Specify LOGGING if you want logging of all tables, indexes, and partitions within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

When an existing tablespace logging attribute is changed by an ALTER TABLESPACE statement, all tables, indexes, and partitions created *after* the statement will have the new default logging attribute (which you can still subsequently override). The logging attributes of existing objects are not changed.

If the tablespace is in FORCE LOGGING mode, then you can specify NOLOGGING in this statement to set the default logging mode of the tablespace to NOLOGGING, but this will not take the tablespace out of FORCE LOGGING mode.

[NO] FORCE LOGGING

Use this clause to put the tablespace in force logging mode or take it out of force logging mode. The database must be open and in READ WRITE mode. Neither of these settings changes the default LOGGING or NOLOGGING mode of the tablespace.

Restriction on FORCE LOGGING: You cannot specify FORCE LOGGING for an undo or a temporary tablespace.

See Also: *Oracle9i Database Administrator's Guide* for information on when to use FORCE LOGGING mode and "Changing Tablespace Logging Attributes: Example" on page 11-110

Examples

Backing Up Tablespaces: Examples The following statement signals to the database that a backup is about to begin:

```
ALTER TABLESPACE tbs_01
    BEGIN BACKUP;
```

The following statement signals to the database that the backup is finished:

```
ALTER TABLESPACE tbs_01
    END BACKUP;
```

Moving and Renaming Tablespaces: Example This example moves and renames a datafile associated with the tbs_01 tablespace from 'diskb:tbs_f5.dat' to 'diska:tbs_f5.dat':

1. Take the tablespace offline using an ALTER TABLESPACE statement with the OFFLINE clause:

   ```
   ALTER TABLESPACE tbs_01 OFFLINE NORMAL;
   ```

2. Copy the file from 'diskb:tbs_f5.dat' to 'diska:tbs_f5.dat' using your operating system's commands.

3. Rename the datafile using the ALTER TABLESPACE statement with the RENAME DATAFILE clause:

   ```
   ALTER TABLESPACE tbs_01
     RENAME DATAFILE 'diskb:tbs_f5.dat'
     TO              'diska:tbs_f5.dat';
   ```

4. Bring the tablespace back online using an ALTER TABLESPACE statement with the ONLINE clause:

   ```
   ALTER TABLESPACE tbs_01 ONLINE;
   ```

Adding a Datafile: Example The following statement adds a datafile to the tablespace. When more space is needed, new extents of size 10 kilobytes will be added up to a maximum of 100 kilobytes:

```
ALTER TABLESPACE tbs_03
```

```
ADD DATAFILE 'tbs_f04.dbf'
SIZE 50K
AUTOEXTEND ON
NEXT 10K
MAXSIZE 100K;
```

Adding an Oracle-managed Datafile: Example The following example adds an Oracle-managed datafile to the `omf_ts1` tablespace (see "Creating Oracle-managed Files: Examples" on page 15-91 for the creation of this tablespace). The new datafile is 100M and is autoextensible with unlimited maximum size:

```
ALTER TABLESPACE omf_ts1 ADD DATAFILE;
```

Changing Tablespace Logging Attributes: Example The following example changes the default logging attribute of a tablespace to `NOLOGGING`:

```
ALTER TABLESPACE tbs_03 NOLOGGING;
```

Altering a tablespace logging attribute has no affect on the logging attributes of the existing schema objects within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Changing Tablespace Extent Allocation: Example The following statement changes the allocation of every extent of `tbs_03` to a multiple of 128K:

```
ALTER TABLESPACE tbs_03 MINIMUM EXTENT 128K;
```

12

SQL Statements: ALTER TRIGGER to COMMIT

This chapter contains the following SQL statements:

- ALTER TRIGGER
- ALTER TYPE
- ALTER USER
- ALTER VIEW
- ANALYZE
- ASSOCIATE STATISTICS
- AUDIT
- CALL
- COMMENT
- COMMIT

ALTER TRIGGER

Purpose

Use the ALTER TRIGGER statement to enable, disable, or compile a database trigger.

Note: This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the CREATE TRIGGER statement with the OR REPLACE keywords.

See Also:

- CREATE TRIGGER on page 15-95 for information on creating a trigger
- DROP TRIGGER on page 17-13 for information on dropping a trigger
- *Oracle9i Database Concepts* for general information on triggers

Prerequisites

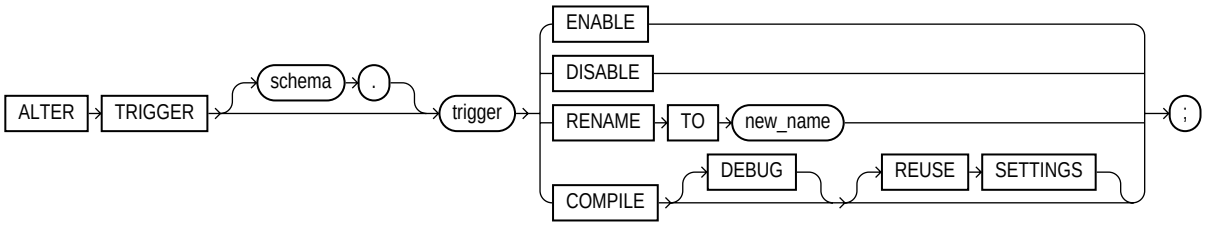
The trigger must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

In addition, to alter a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

See Also: CREATE TRIGGER on page 15-95 for more information on triggers based on DATABASE triggers

Syntax

alter_trigger::=



Keywords and Parameters

schema

Specify the schema containing the trigger. If you omit *schema*, then Oracle assumes the trigger is in your own schema.

trigger

Specify the name of the trigger to be altered.

ENABLE Clause

Specify ENABLE to enable the trigger. You can also use the ENABLE ALL TRIGGERS clause of ALTER TABLE to enable all triggers associated with a table.

> **See Also:** ALTER TABLE on page 11-2 and "Enabling Triggers: Example" on page 12-5

DISABLE Clause

Specify DISABLE to disable the trigger. You can also use the DISABLE ALL TRIGGERS clause of ALTER TABLE to disable all triggers associated with a table.

> **See Also:** ALTER TABLE on page 11-2 and "Disabling Triggers: Example" on page 12-4

RENAME Clause

Specify RENAME TO *new_name* to rename the trigger. Oracle renames the trigger and leaves it in the same state it was in before being renamed.

Note: When you rename a trigger, Oracle rebuilds the remembered source of the trigger in the USER_SOURCE, ALL_SOURCE, and DBA_SOURCE data dictionary views. As a result, comments and formatting may change in the TEXT column of those views even though the trigger source did not change.

COMPILE Clause

Specify COMPILE to explicitly compile the trigger, whether it is valid or invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Oracle first recompiles objects upon which the trigger depends, if any of these objects are invalid. If Oracle recompiles the trigger successfully, then the trigger becomes valid.

During recompilation, Oracle drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the REUSE SETTINGS clause.

If recompiling the trigger results in compilation errors, then Oracle returns an error and the trigger remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

DEBUG Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information on debugging procedures
- *Oracle9i Database Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects

REUSE SETTINGS Specify REUSE SETTINGS to prevent Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation.

If you specify both DEBUG and REUSE SETTINGS, Oracle sets the persistently stored value of the PLSQL_COMPILER_FLAGS parameter to INTERPRETED, DEBUG. No other compiler switch values are changed.

See Also: *PL/SQL User's Guide and Reference* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on the interaction of the PLSQL_COMPILER_FLAGS parameter with the COMPILE clause

Examples

Disabling Triggers: Example The sample schema hr has a trigger named update_job_history created on the employees table. The trigger is fired whenever an UPDATE statement changes an employee's job_id. The trigger inserts

into the job_history table a row that contains the employee's ID, begin and end date of the last job, and the job ID and department.

When this trigger is created, Oracle enables it automatically. You can subsequently disable the trigger with the following statement:

```
ALTER TRIGGER update_job_history DISABLE;
```

When the trigger is disabled, Oracle does not fire the trigger when an UPDATE statement changes an employee's job.

Enabling Triggers: Example After disabling the trigger, you can subsequently enable it with the following statement:

```
ALTER TRIGGER update_job_history ENABLE;
```

After you reenable the trigger, Oracle fires the trigger whenever an employee's job changes as a result of an UPDATE statement. If an employee's job is updated while the trigger is disabled, then Oracle does not automatically fire the trigger for this employee until another transaction changes the job_id again.

ALTER TYPE

Purpose

Use the ALTER TYPE statement to add or drop member attributes or methods. You can change the existing properties (FINAL or INSTANTIABLE) of an object type, and you can modify the scalar attributes of the type.

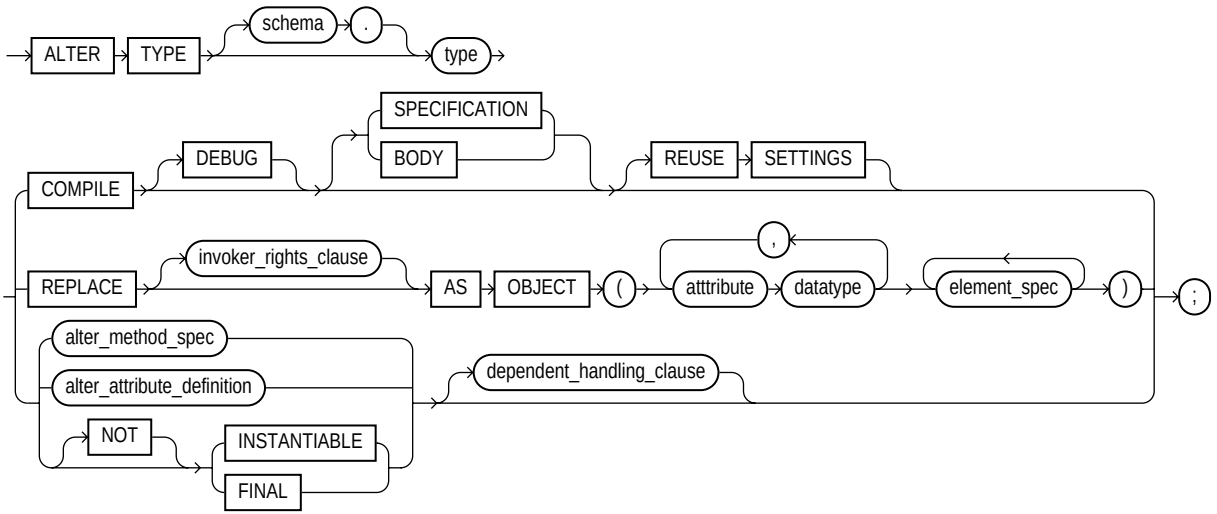
You can also use this statement to recompile the specification or body of the type or to change the specification of an object type by adding new object member subprogram specifications.

Prerequisites

The object type must be in your own schema and you must have CREATE TYPE or CREATE ANY TYPE system privilege, or you must have ALTER ANY TYPE system privileges.

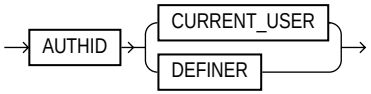
Syntax

alter_type::=

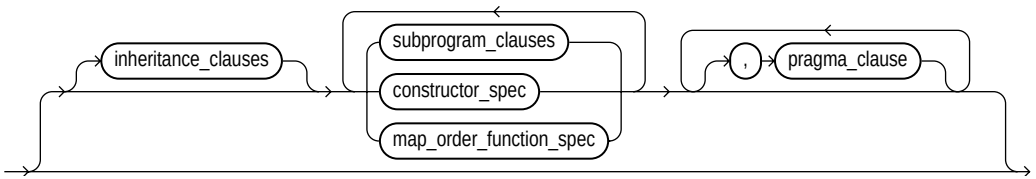


(*invoker_rights_clause::=* on page 12-7, *alter_method_spec::=* on page 12-8, *alter_attribute_definition::=* on page 12-9, *dependent_handling_clause::=* on page 12-9)

invoker_rights_clause::=

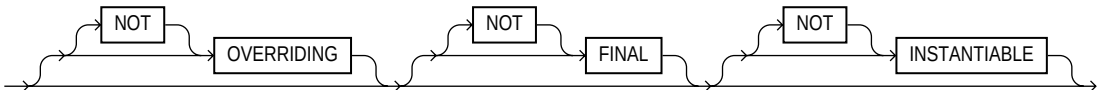


element_spec::=

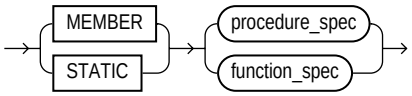


(*inheritance_clauses::=* on page 12-7, *subprogram_clauses::=* on page 12-7, *constructor_spec::=* on page 12-8, *map_order_function_spec::=* on page 12-8, *pragma_clause::=* on page 12-8)

inheritance_clauses::=

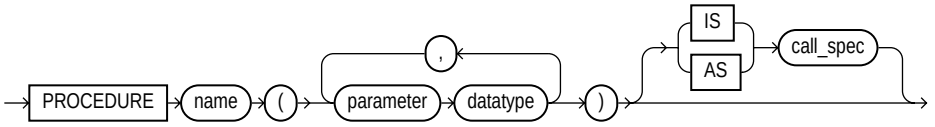


subprogram_clauses::=

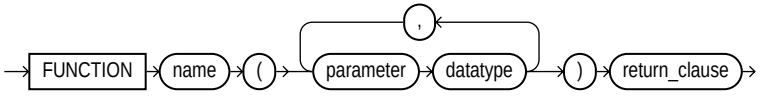


(*procedure_spec::=* on page 12-7, *function_spec::=* on page 12-7)

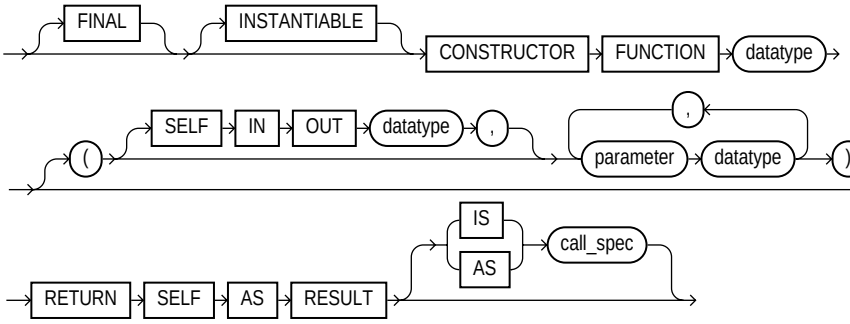
procedure_spec::=



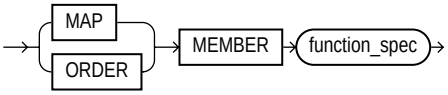
function_spec::=



constructor_spec::=

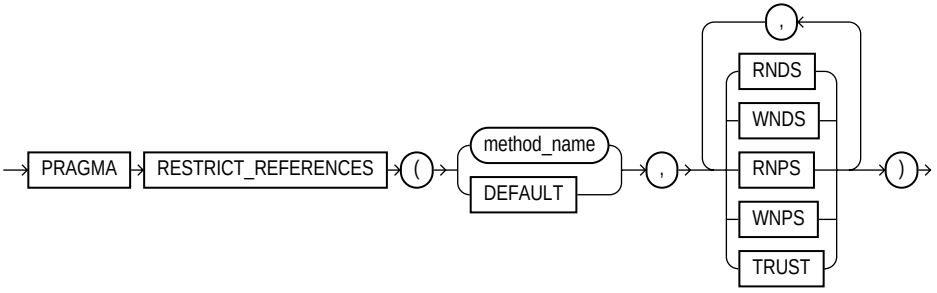


map_order_function_spec::=

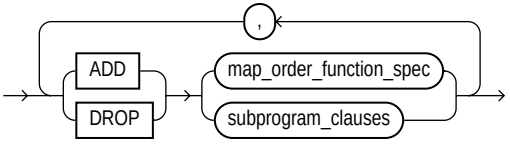


(*function_spec::=* on page 12-7)

pragma_clause::=

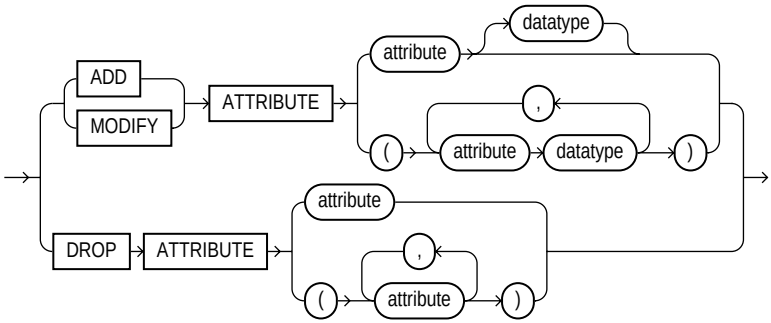


alter_method_spec::=

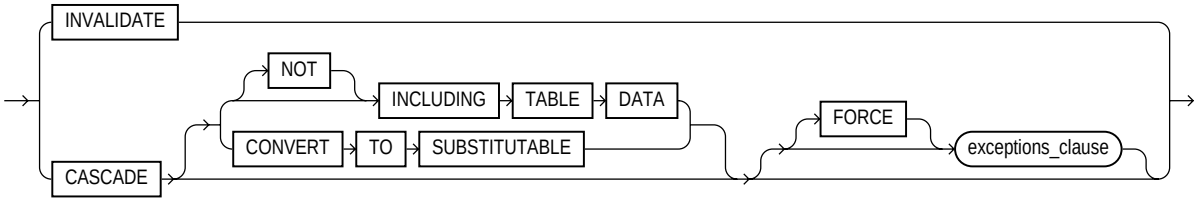


(*map_order_function_spec::=* on page 12-8, *subprogram_clauses::=* on page 12-7)

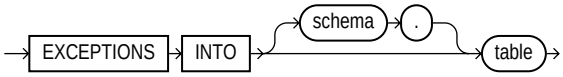
alter_attribute_definition::=



dependent_handling_clause::=



exceptions_clause::=



Keywords and Parameters

schema

Specify the schema that contains the type. If you omit *schema*, then Oracle assumes the type is in your current schema.

type

Specify the name of an object type, a nested table type, or a rowid type.

COMPILE Clause

Specify `COMPILE` to compile the object type specification and body. This is the default if neither `SPECIFICATION` nor `BODY` is specified.

During recompilation, Oracle drops all persistent compiler switch settings, retrieves them again from the session, and stores them at the end of compilation. To avoid this process, specify the REUSE SETTINGS clause.

If recompiling the type results in compilation errors, then Oracle returns an error and the type remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

> **See Also:** "Recompiling a Type: Example" on page 12-19 and "Recompiling a Type Specification: Example" on page 12-19

DEBUG Specify DEBUG to instruct the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

SPECIFICATION Specify SPECIFICATION to compile only the object type specification.

BODY Specify BODY to compile only the object type body.

REUSE SETTINGS Specify REUSE SETTINGS to prevent Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation.

If you specify both DEBUG and REUSE SETTINGS, Oracle sets the persistently stored value of the PLSQL_COMPILER_FLAGS parameter to INTERPRETED, DEBUG. No other compiler switch values are changed.

> **See Also:** *PL/SQL User's Guide and Reference* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on the interaction of the PLSQL_COMPILER_FLAGS parameter with the COMPILE clause

REPLACE AS OBJECT Clause

The REPLACE AS OBJECT clause lets you add new member subprogram specifications. This clause is valid only for object types, not for nested table or varray types.

attribute

Specify an object attribute name. Attributes are data items with a name and a type specifier that form the structure of the object.

element_spec

Specify the elements of the redefined object.

inheritance_clauses As part of the *element_spec*, the *inheritance_clauses* let you specify the relationship between super- and subtypes.

OVERRIDING This clause is valid only for MEMBER methods. Specify OVERRIDING to indicate that this method overrides a MEMBER method defined in the supertype. This keyword is required if the method redefines a supertype method. NOT OVERRIDING is the default.

Restriction: The OVERRIDING clause is not valid for a STATIC method or for a SQLJ object type.

FINAL Specify FINAL to indicate that this method cannot be overridden by any subtype of this type. The default is NOT FINAL.

NOT INSTANTIABLE Specify NOT INSTANTIABLE if the type does not provide an implementation for this method. By default all methods are INSTANTIABLE.

Restriction on NOT INSTANTIABLE: If you specify NOT INSTANTIABLE, you cannot specify FINAL or STATIC.

subprogram_clauses The MEMBER and STATIC clauses let you specify a function or procedure subprogram associated with the object type which is referenced as an attribute.

You must specify a corresponding method body in the object type body for each procedure or function specification.

See Also:

- CREATE TYPE on page 16-3 for a description of the difference between member and static methods, and for examples
- *PL/SQL User's Guide and Reference* for information about overloading subprogram names within a package
- CREATE TYPE BODY on page 16-25

procedure_spec Enter the specification of a procedure subprogram.

function_spec Enter the specification of a function subprogram.

pragma_clause The *pragma_clause* is a complier directive that denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

Oracle Corporation recommends that you avoid using this clause unless you must do so for backward compatibility of your applications. This clause has been deprecated, because beginning with Oracle9*i*, Oracle runs purity checks at run time.

Restriction on the *pragma_clause*: The *pragma_clause* is not valid when dropping a method.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals*

map_order_function_spec You can declare either a MAP method or an ORDER method, but not both. However, a subtype can override a MAP method if the supertype defines a NOT FINAL MAP method. If you declare either method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types.

> **See Also:** "Object Values" on page 2-48for more information about object value comparisons

- For MAP, specify a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. Oracle uses the ordering for comparison conditions and ORDER BY clauses.

  If the argument to the MAP method is null, then the MAP method returns null and the method is not invoked.

  An object specification can contain only one MAP method, which must be a function. The result type must be a predefined SQL scalar type, and the MAP function can have no arguments other than the implicit SELF argument.

  Note: If *type* will be referenced in queries involving sorts (through ORDER BY, GROUP BY, DISTINCT, or UNION clauses) or joins, and you want those queries to be parallelized, then you must specify a MAP member function.

A subtype cannot define a new MAP method. However, it can override an inherited MAP method.

- For ORDER, specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, zero, or positive indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

  If either argument to the ORDER method is null, then the ORDER method returns null and the method is not invoked.

  When instances of the same object type definition are compared in an ORDER BY clause, the ORDER method function is invoked.

  An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

  A subtype cannot define an ORDER method, nor can it override an inherited ORDER method.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the member functions and procedures of the object type execute with the privileges and in the schema of the user who owns the object type or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding type body as well.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

Restriction on the *invoker_rights_clause***:** You can specify this clause only for an object type, not for a nested table or varray type.

AUTHID CURRENT_USER Clause Specify CURRENT_USER if you want the member functions and procedures of the object type to execute with the privileges of CURRENT_USER. This clause creates an **invoker-rights type**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the type resides.

Note: You must specify this clause to maintain invoker-rights status for the type if you created it with this status. Otherwise the status will revert to definer rights.

AUTHID DEFINER Clause Specify DEFINER if you want the member functions and procedures of the object type to execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default.

See Also:

- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *PL/SQL User's Guide and Reference*

alter_method_spec

The *alter_method_spec* lets you add a method to or drop a method from *type*. Oracle disables any function-based indexes that depend on the type.

In one ALTER TYPE statement you can add or drop multiple methods, but you can reference each method only once.

ADD When you add a method, its name must not conflict with any existing attributes in its type hierarchy.

See Also: "Adding a Member Function: Example" on page 12-18

DROP When you drop a method, Oracle removes the method from the target type.

Restriction on DROP: You cannot drop from a subtype a method inherited from its supertype. Instead you must drop the method from the supertype.

subprogram_clauses The MEMBER and STATIC clauses let you add a procedure subprogram to or drop it from the object type.

Restriction on *subprogram_clauses*: You cannot define a STATIC method on a subtype that redefines a MEMBER method in its supertype, or vice versa.

> **See Also:** the description of the *subprogram_clauses* in CREATE TYPE on page 16-12

map_order_function_spec If you declare either a MAP or ORDER method, then you can compare object instances in SQL.

Restriction on *map_order_function_spec*: You cannot add an ORDER method to a subtype.

> **See Also:** the description of *constructor_spec* in CREATE TYPE on page 16-16

alter_attribute_definition

The *alter_attribute_definition* clause lets you add, drop, or modify an attribute of an object type. In one ALTER TYPE statement, you can add, drop, or modify multiple member attributes or methods, but you can reference each attribute or method only once.

ADD ATTRIBUTE The name of the new attribute must not conflict with existing attributes or methods in the type hierarchy. Oracle adds the new attribute to the end of the locally defined attribute list.

Note: If you add the attribute to a supertype, then it is inherited by all of its subtypes. In subtypes, inherited attributes always precede declared attributes. Therefore, you may need to update the mappings of the implicitly altered subtypes after adding an attribute to a supertype.

> **See Also:** "Adding a Collection Attribute: Example" on page 12-19

DROP ATTRIBUTE When you drop an attribute from a type, Oracle drops the column corresponding to the dropped attribute as well as any indexes, statistics, and constraints referencing the dropped attribute.

You need not specify the datatype of the attribute you are dropping.

Restrictions on dropping attributes:

- You cannot drop an attribute inherited from a supertype. Instead you must drop the attribute from the supertype.

- You cannot drop an attribute that is part of a partitioning, subpartitioning, or cluster key.
- You cannot drop an attribute of a primary-key-based object identifier of an object table or a primary key of an index-organized table.
- You cannot drop all of the attributes of a root type. Instead you must drop the type. However, you can drop all of the locally declared attributes of a subtype.

MODIFY ATTRIBUTE This clause lets you modify the datatype of an existing scalar attribute. For example, you can increase the length of a VARCHAR2 or RAW attribute, or you can increase the precision or scale of a numeric attribute.

Restriction on modifying an attribute: You cannot expand the size of an attribute referenced in a function-based index, domain index, or cluster key.

[NOT] FINAL

Use the [NOT] FINAL clause to indicate whether any further subtypes can be created for this type:

- Specify FINAL if no further subtypes can be created for this type.
- Specify NOT FINAL if further subtypes can be created under this type.

If you change the property between FINAL and NOT FINAL, then you must specify the CASCADE clause of the *dependent_handling_clause* to convert data in dependent columns and tables. You cannot defer data conversion with CASCADE NOT INCLUDING TABLE DATA.

- If you change a type from NOT FINAL to FINAL, then you must specify CASCADE [INCLUDING TABLE DATA].
- If you change a type from FINAL to NOT FINAL:
  - Specify CASCADE INCLUDING TABLE DATA if you want to create new substitutable tables and columns of that type, but you are not concerned about the substitutability of the existing dependent tables and columns. Oracle marks all existing dependent columns and tables NOT SUBSTITUTABLE AT ALL LEVELS, so you cannot insert the new subtype instances of the altered type into these existing columns and tables.
  - Specify CASCADE CONVERT TO SUBSTITUTABLE if you want to create new substitutable tables and columns of the type and also store new subtype instances of the altered type in existing dependent tables and columns. Oracle marks all existing dependent columns and tables SUBSTITUTABLE

AT ALL LEVELS except those that are explicitly marked NOT SUBSTITUTABLE AT ALL LEVELS.

See Also: *Oracle9i Application Developer's Guide - Object-Relational Features* for a full discussion of object type evolution

Restriction on FINAL: You cannot change a user-defined type from NOT FINAL to FINAL if the type has any subtypes.

[NOT] INSTANTIABLE

Use the [NOT] INSTANTIABLE clause to indicate whether any object instances of this type can be constructed:

- Specify INSTANTIABLE if object instances of this type can be constructed.
- Specify NOT INSTANTIABLE if no constructor (default or user-defined) exists for this object type. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement).

Restriction on NOT INSTANTIABLE: You cannot change a user-defined type from INSTANTIABLE to NOT INSTANTIABLE if the type has any table dependents.

dependent_handling_clause

The *dependent_handling_clause* lets you instruct Oracle how to handle objects that are dependent on the modified type. If this clause is not specified, then the ALTER TYPE statement will abort if the target type has any dependent type or table.

INVALIDATE Clause

Specify INVALIDATE to invalidate all dependent objects without any checking mechanism.

Note: Because Oracle does not validate the type change, you should use this clause with caution. For example, if you drop an attribute that is a partitioning or cluster key, then you will be unable to write to the table.

CASCADE Clause

Specify the CASCADE clause if you want to propagate the type change to dependent types and tables. Oracle aborts the statement if any errors are found in the dependent types or tables unless you also specify FORCE.

INCLUDING TABLE DATA. Specify INCLUDING TABLE DATA to convert data stored in all user-defined columns to the most recent version of the column's type. This is the default.

> **Note:** You must specify this clause if your column data is in Oracle8 release 8.0 image format. This clause is also required if the type property is being changed between FINAL and NOT FINAL

- For each attribute added to the column's type, Oracle adds a new attribute to the data and initializes it to NULL.
- For each attribute dropped from the referenced type, Oracle removes the corresponding attribute data from each row in the table.

When you specify INCLUDING TABLE DATA, all of the tablespaces containing the table's data must be in read/write mode.

If you specify NOT INCLUDING TABLE DATA, then Oracle upgrades the metadata of the column to reflect the changes to the type, but does not scan the dependent column and update the data as part of this ALTER TYPE statement. However, the dependent column data remains accessible and the results of subsequent queries of the data will reflect the type modifications.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for more information on the implications of not including table data when modifying type attribute

FORCE. Specify FORCE if you want Oracle to ignore the errors from dependent tables and indexes and log all errors in the specified exception table. The exception table must already have been created by executing the DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE procedure.

Examples

Adding a Member Function: Example. The following example uses the data_typ object type, which was created in "Object Type Examples" on page 16-19. A method

is added to data_typ and its type body is modified to correspond. The date formats are consistent with the order_date column of the oe.orders sample table:

```
ALTER TYPE data_typ
   ADD MEMBER FUNCTION qtr(der_qtr DATE)
   RETURN CHAR CASCADE;

CREATE OR REPLACE TYPE BODY data_typ IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
  BEGIN
  RETURN (year + invent);
  END;
     MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
     BEGIN
       IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
         RETURN 'FIRST';
       ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
         RETURN 'SECOND';
       ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
         RETURN 'THIRD';
       ELSE
         RETURN 'FOURTH';
       END IF;
     END;
   END;
/
```

Adding a Collection Attribute: Example. The following example adds the phone_list_typ varray from the sample oe schema to the customer_address_typ object column of the customers table:

```
ALTER TYPE cust_address_typ
   ADD ATTTRIBUTE (phone phone_list_typ) CASCADE;
```

Recompiling a Type: Example. The following example recompiles type customer_address_typ:

```
ALTER TYPE customer_address_typ COMPILE;
```

Recompiling a Type Specification: Example. The following example compiles the type specification of link2.

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);
/
```

```
CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);
/

CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS
      BEGIN
         dbms_output.put_line(c1);
         RETURN c1;
      END;
   END;
/
```

In the following example, both the specification and body of `link2` are invalidated.

```
ALTER TYPE link1 ADD ATTRIBUTE (b NUMBER) INVALIDATE;
```

You must recompile the type by recompiling the specification and body in separate statements:

```
ALTER TYPE link2 COMPILE SPECIFICATION;

ALTER TYPE link2 COMPILE BODY;
```

Alternatively, you can compile both specification and body at the same time:

```
ALTER TYPE link2 COMPILE;
```

ALTER USER

Purpose

Use the ALTER USER statement:

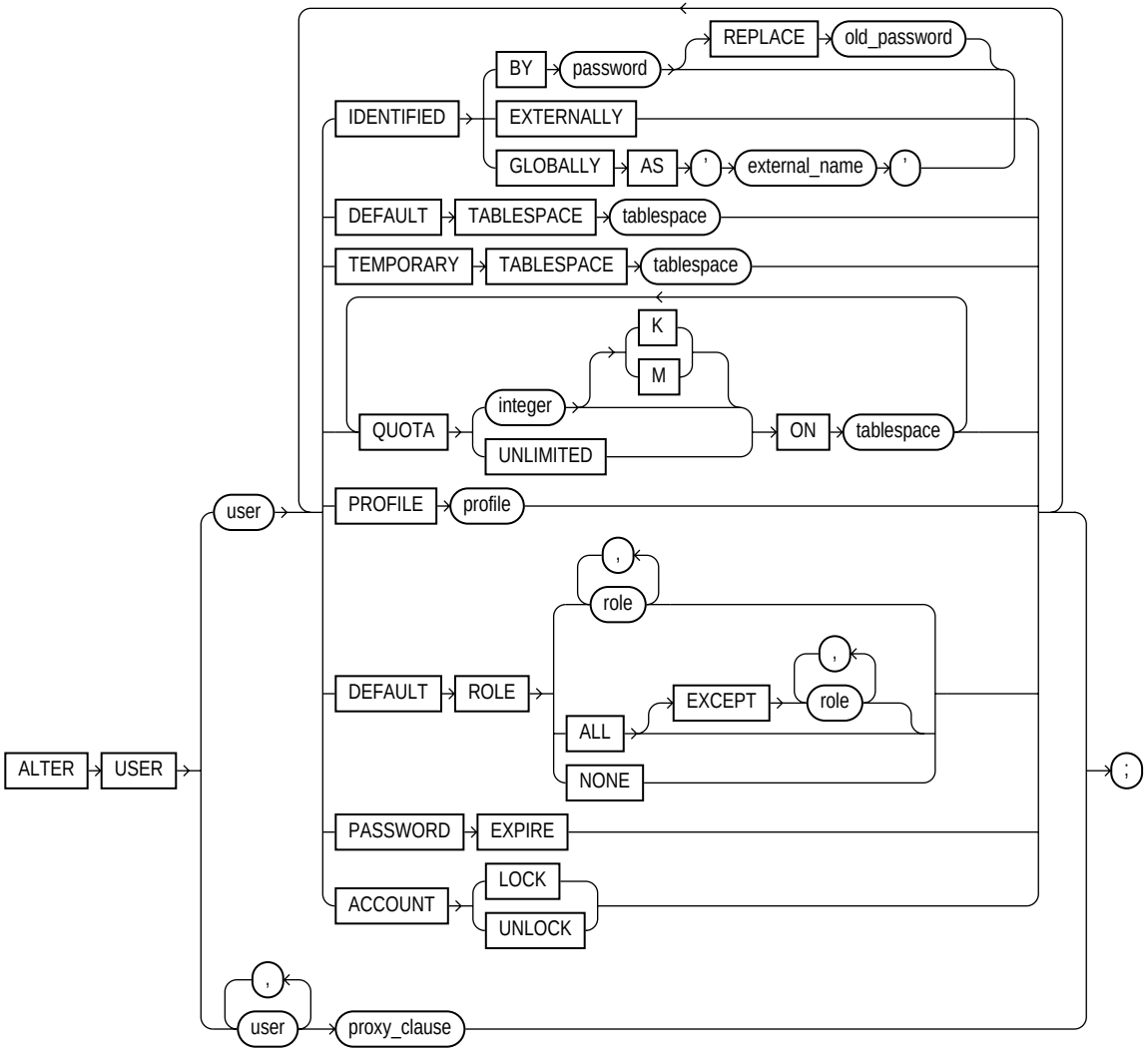
- To change the authentication or database resource characteristics of a database user.
- To permit a proxy server to connect as a client without authentication.

Prerequisites

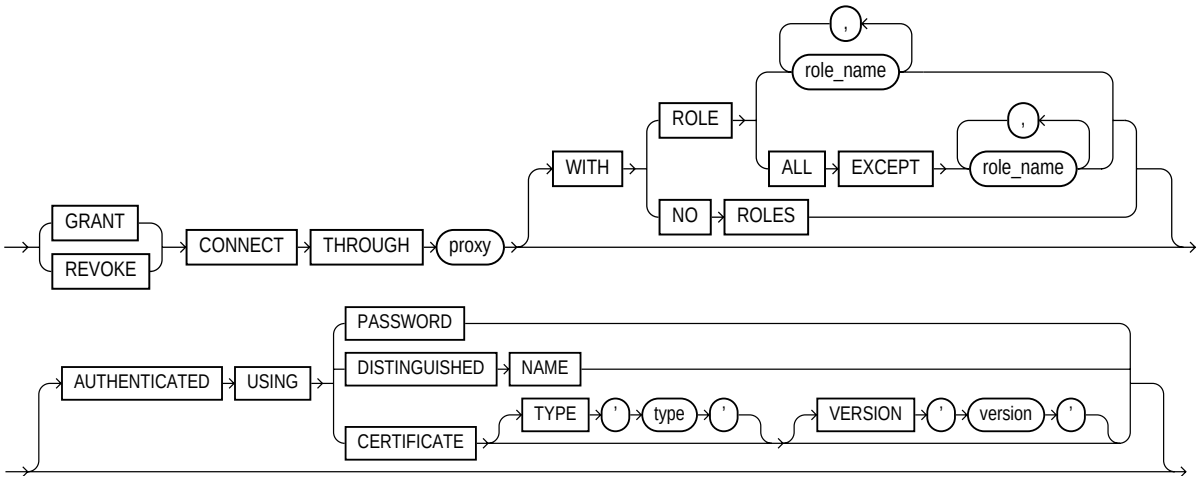
You must have the ALTER USER system privilege. However, you can change your own password without this privilege.

Syntax

alter_user::=



proxy_clause::=



Keywords and Parameters

The keywords and parameters described in this section are unique to ALTER USER or have different semantics than they have in CREATE USER. All the remaining keywords and parameters in the ALTER USER statement have the same meaning as in the CREATE USER statement.

Note: Oracle Corporation recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform. Please refer to *Oracle9i Database Administrator's Guide* for more information about this recommendation.

See Also:

- CREATE USER on page 16-32 for information on the keywords and parameters
- CREATE PROFILE on page 14-69 for information on assigning limits on database resources to a user

IDENTIFIED Clause

- Specify BY *password* to specify a new password for the user.

Note: Oracle expects a different timestamp for each resetting of a particular password. If you reset one password multiple times within one second (for example, by cycling through a set of passwords using a script), then Oracle may return an error message that the password cannot be reused. For this reason, Oracle Corporation recommends that you avoid using scripts to reset passwords.

If the password complexity verification function is turned on (that is, if the UTLPWDMG.SQL script has been run), then you must specify the REPLACE clause if you are changing your own existing password. You can omit the REPLACE clause if you are setting your own password for the first time or if you have the ALTER USER privilege and are changing the password of another user.

Note: Oracle does not check the old password, even if you provide it in the REPLACE clause, unless you are changing your own existing password. If such a check is important in other cases (for example, when a privileged user changes another user's password), then ensure that the password complexity verification function prohibits password changes in which the old password is null, or use the OCIPasswordChange() call instead of ALTER USER. For more information, see *Oracle Call Interface Programmer's Guide*.

See Also: *Oracle9i Database Administrator's Guide* for information on the password complexity verification function

- Specify GLOBALLY AS '*external_name*' to indicate that the user must be authenticated by way of an LDAP V3 compliant directory service such as Oracle Internet Directory.

  You can change a user's access verification method to IDENTIFIED GLOBALLY AS '*external_name*' only if all external roles granted directly to the user are revoked.

  You can change a user created as IDENTIFIED GLOBALLY AS '*external_name*' to IDENTIFIED BY *password* or IDENTIFIED EXTERNALLY.

See Also: CREATE USER on page 16-32, "Changing User Identification: Example" on page 12-27, and "Changing User Authentication: Examples" on page 12-27

TEMPORARY TABLESPACE Clause

The tablespace you assign or reassign as the user's temporary tablespace must be a temporary tablespace and must have a standard block size.

DEFAULT ROLE Clause

Specify the roles granted by default to the user at logon. This clause can contain only roles that have been granted directly to the user with a `GRANT` statement. You cannot use the `DEFAULT ROLE` clause to enable:

- Roles not granted to the user
- Roles granted through other roles
- Roles managed by an external service (such as the operating system), or by the Oracle Internet Directory

Oracle enables default roles at logon without requiring the user to specify their passwords.

See Also: CREATE ROLE on page 14-77

proxy_clause

The *`proxy_clause`* lets you control the ability of a **proxy** (an application or application server) to connect as the specified database or enterprise user and to activate all, some, or none of the user's roles.

Note: The *`proxy_clause`* provides several varieties of proxy authentication of database and enterprise users. For information on proxy authentication of application users, see *Oracle9i Application Developer's Guide - Fundamentals*.

See Also: *Oracle9i Database Concepts* for more information on proxies and their use of the database and "Proxy Users: Examples" on page 12-28

GRANT | REVOKE

Specify GRANT to allow the connection. Specify REVOKE to prohibit the connection.

CONNECT THROUGH Clause

Identify the proxy connecting to Oracle. Oracle expects the proxy to authenticate the user unless you specify the AUTHENTICATED USING clause.

WITH ROLE WITH ROLE *role_name* permits the proxy to connect as the specified user and to activate only the roles that are specified by *role_name*.

WITH ROLE ALL EXCEPT WITH ROLE ALL EXCEPT *role_name* permits the proxy to connect as the specified user and to activate all roles associated with that user except those specified by *role_name*.

WITH NO ROLES WITH NO ROLES permits the proxy to connect as the specified user, but prohibits the proxy from activating any of that user's roles after connecting.

If you do not specify any of these WITH clauses, then Oracle activates all roles granted to the specified user automatically.

AUTHENTICATED USING

Specify the AUTHENTICATED USING clause if you want proxy authentication to be handled by a source other than the proxy. This clause is relevant only as part of a GRANT CONNECT THROUGH *proxy* clause.

PASSWORD Specify PASSWORD if you want the proxy to present the database password of the user for authentication. The proxy relies on the database to authenticate the user based on the password.

DISTINGUISHED NAME Specify DISTINGUISHED NAME to allow the proxy to act as the globally identified user indicated by the distinguished name.

CERTIFICATE Specify CERTIFICATE to allow the proxy to act as the globally identified user whose distinguished name is contained in the certificate.

In both the DISTINGUISHED NAME and CERTIFICATE cases, the proxy has already authenticated and is acting on behalf of a global database user.

- For *type*, specify the type of certificate to be presented. If you do not specify *type*, then the default is 'X.509'.

- For *version*, specify the version of the certificate that is to be presented. If you do not specify *version*, then the default is '3'.

Restriction on CERTIFICATE: You cannot specify this clause as part of a REVOKE CONNECT THROUGH *proxy* clause.

See Also:

- *Oracle9i Security Overview* for an overview of database security
- *Oracle9i Database Administrator's Guide* and *Oracle9i Application Developer's Guide - Fundamentals* for information on middle-tier systems and proxy authentication

Examples

Changing User Identification: Example The following statement changes the password of the user sidney (created in "Creating a Database User: Example" on page 16-37) second_2nd_pwd and default tablespace to the tablespace demo:

```
ALTER USER sidney
    IDENTIFIED BY second_2nd_pwd
    DEFAULT TABLESPACE demo;
```

The following statement assigns the new_profile profile "Creating a Profile: Example" on page 14-74) to the sample user sh:

```
ALTER USER sh
    PROFILE new_profile;
```

In subsequent sessions, sh is restricted by limits in the new_profile profile.

The following statement makes all roles granted directly to sh default roles, except the dw_manager role:

```
ALTER USER sh
    DEFAULT ROLE ALL EXCEPT dw_manager;
```

At the beginning of sh's next session, Oracle enables all roles granted directly to sh except the dw_manager role.

Changing User Authentication: Examples The following statement changes the authentication mechanism of user app_user1(created in "Creating a Database User: Example" on page 16-37) :

```
ALTER USER app_user1 IDENTIFIED GLOBALLY AS 'CN=tom,O=oracle,C=US';
```

The following statement causes user sidney's password to expire:

```
ALTER USER sidney PASSWORD EXPIRE;
```

If you cause a database user's password to expire with PASSWORD EXPIRE, then the user (or the DBA) must change the password before attempting to log in to the database following the expiration. However, tools such as SQL*Plus allow the user to change the password on the first attempted login following the expiration.

Proxy Users: Examples The following statement alters the user app_user. The example permits the app_user to connect through the proxy user sh. The example also allows app_user to enable its warehouse_user role (created in "Creating a Role: Example" on page 14-79) when connected through the proxy sh:

```
ALTER USER app_user1
   GRANT CONNECT THROUGH sh
   WITH ROLE warehouse_user;
```

Note: To show basic syntax, this example uses the sample database Sales History user (sh) as the proxy. Normally a proxy user would be an application server or middle-tier entity. For information on creating the interface between an application user and a database by way of an application server, please refer to *Oracle Call Interface Programmer's Guide.*

See Also:

- "Creating External Database Users: Examples" on page 16-38 to see how to create the app_user user
- "Creating a Role: Example" on page 14-79 to see how to create the dw_user role

The following statement takes away the right of user app_user to connect through the proxy user sh:

```
ALTER USER app_user1 REVOKE CONNECT THROUGH sh;
```

The following hypothetical examples show other methods of proxy authentication:

```
ALTER USER grant GRANT CONNECT THROUGH OAS1
   AUTHENTICATED USING PASSWORD;
```

```
ALTER USER green GRANT CONNECT THROUGH WebDB
   AUTHENTICATED USING DISTINGUISHED NAME;

ALTER USER brown GRANT CONNECT THROUGH WebDB
   AUTHENTICATED USING CERTIFICATE TYPE 'X.509' VERSION '3';
```

ALTER VIEW

Purpose

Use the ALTER VIEW statement to explicitly recompile a view that is invalid. Explicit recompilation lets you locate recompilation errors before run time. You may want to recompile a view explicitly after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it.

You can also use ALTER VIEW to define, modify, or drop view constraints.

When you issue an ALTER VIEW statement, Oracle recompiles the view regardless of whether it is valid or invalid. Oracle also invalidates any local objects that depend on the view.

Notes:

- This statement does not change the definition of an existing view. To redefine a view, you must use CREATE VIEW with the OR REPLACE keywords.
- If you alter a view that is referenced by one or more materialized views, then those materialized views are invalidated. Invalid materialized views cannot be used by query rewrite and cannot be refreshed.

See Also:

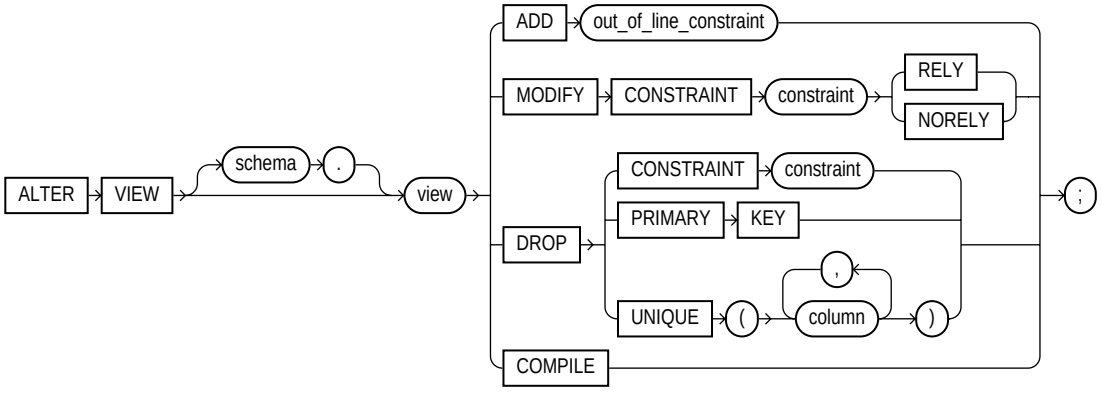
- CREATE VIEW on page 16-39 for information on redefining a view
- ALTER MATERIALIZED VIEW on page 9-92 for information on revalidating an invalid materialized view
- *Oracle9i Data Warehousing Guide* for general information on data warehouses
- *Oracle9i Database Concepts* for more about dependencies among schema objects

Prerequisites

The view must be in your own schema or you must have ALTER ANY TABLE system privilege.

Syntax

alter_view::=



(*out_of_line_constraint::=* on page 7-7—part of *constraints* syntax)

Keywords and Parameters

schema

Specify the schema containing the view. If you omit *schema*, then Oracle assumes the view is in your own schema.

view

Specify the name of the view to be recompiled.

ADD Clause

Use the ADD clause to add a constraint to *view*.

> **See Also:** *constraints* on page 7-5 for information on view constraints and their restrictions

MODIFY CONSTRAINT Clause

Use the MODIFY CONSTRAINT clause to change the RELY or NORELY setting of an existing view constraint.

Restriction on modifying constraints: You cannot change the setting of a unique or primary key constraint if it is part of a referential integrity constraint without dropping the foreign key or changing its setting to match that of *view*.

> **See Also:** "RELY Clause" on page 7-22 for information on the uses of the RELY and NORELY settings

DROP Clause

Use the DROP clause to drop an existing view constraint.

Restriction on dropping constraints: You cannot drop a unique or primary key constraint if it is part of a referential integrity constraint on a view.

COMPILE

The COMPILE keyword is required. It directs Oracle to recompile the view.

Example

Altering a View: Example To recompile the view customer_ro (created in "Creating a Read-Only View: Example" on page 16-52), issue the following statement:

```
ALTER VIEW customer_ro
    COMPILE;
```

If Oracle encounters no compilation errors while recompiling customer_ro, then customer_ro becomes valid. If recompiling results in compilation errors, then Oracle returns an error and customer_ro remains invalid.

Oracle also invalidates all dependent objects. These objects include any procedures, functions, package bodies, and views that reference customer_ro. If you subsequently reference one of these objects without first explicitly recompiling it, then Oracle recompiles it implicitly at run time.

ANALYZE

Purpose

Use the ANALYZE statement to collect non-optimizer statistics, for example, to:

- Collect or delete statistics about an index or index partition, table or table partition, index-organized table, cluster, or scalar object attribute.
- Validate the structure of an index or index partition, table or table partition, index-organized table, cluster, or object reference (REF).
- Identify migrated and chained rows of a table or cluster.

Note: Oracle Corporation strongly recommends that you use the DBMS_STATS package rather than ANALYZE to collect optimizer statistics. That package lets you collect statistics in parallel, collect global statistics for partitioned objects, and fine tune your statistics collection in other ways. Further, the cost-based optimizer, which depends upon statistics, will eventually use only statistics that have been collected by DBMS_STATS. See *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information on this package.

However, you must use the ANALYZE statement (rather than DBMS_STATS) for statistics collection not related to the cost-based optimizer, such as:

- To use the VALIDATE or LIST CHAINED ROWS clauses
- To collect information on freelist blocks

Prerequisites

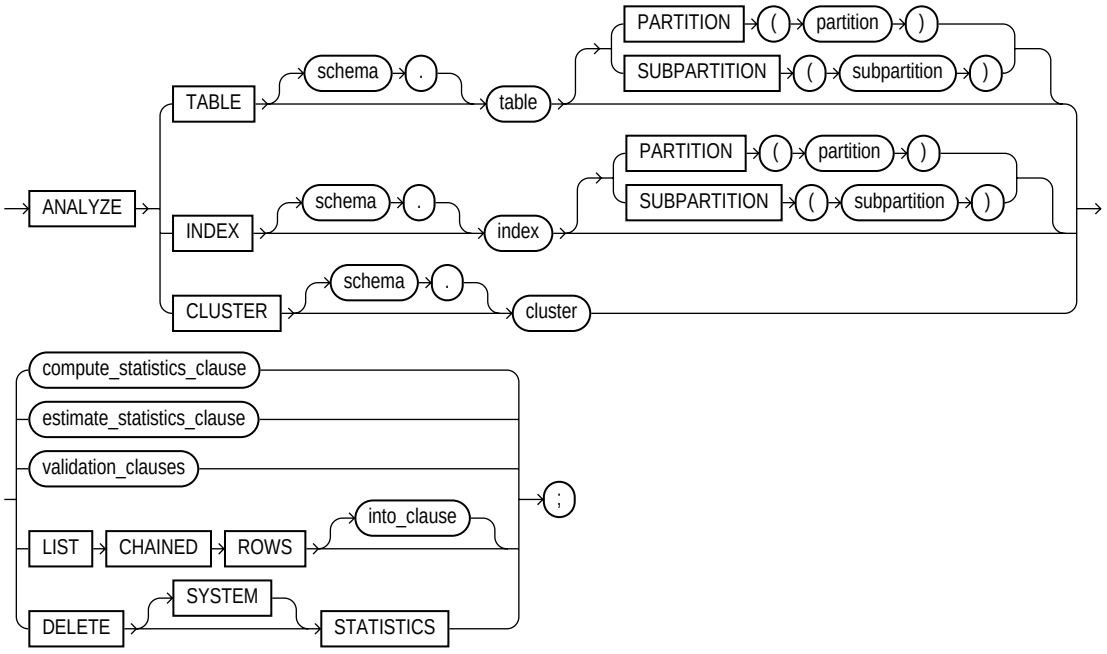
The schema object to be analyzed must be local, and it must be in your own schema or you must have the ANALYZE ANY system privilege.

If you want to list chained rows of a table or cluster into a list table, then the list table must be in your own schema, or you must have INSERT privilege on the list table, or you must have INSERT ANY TABLE system privilege.

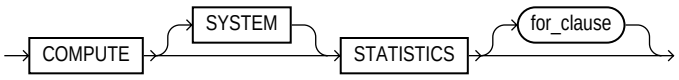
If you want to validate a partitioned table, then you must have INSERT privilege on the table into which you list analyzed rowids, or you must have INSERT ANY TABLE system privilege.

Syntax

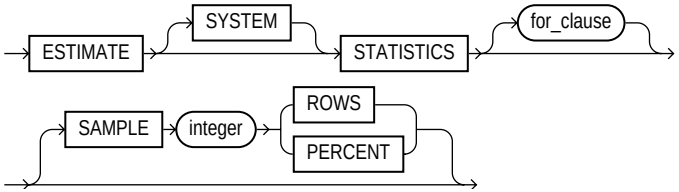
analyze::=



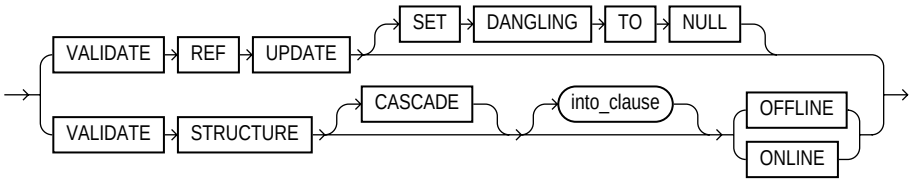
compute_statistics_clause::=



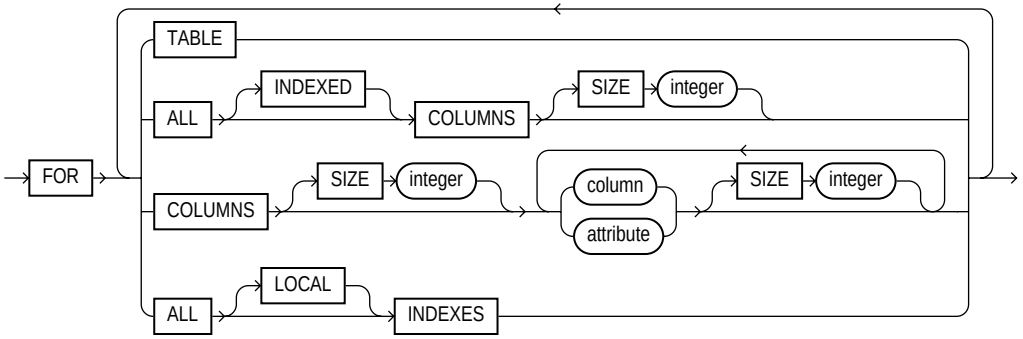
estimate_statistics_clause::=



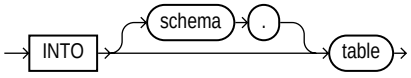
validation_clauses::=



for_clause::=



into_clause::=



Keywords and Parameters

schema

Specify the schema containing the index, table, or cluster. If you omit *schema*, then Oracle assumes the index, table, or cluster is in your own schema.

INDEX *index*

Specify an index to be analyzed (if no *for_clause* is used).

Oracle collects the following statistics for an index. Statistics marked with an asterisk are always computed exactly. For conventional indexes, the statistics appear in the data dictionary views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES in the columns in parentheses.

- *Depth of the index from its root block to its leaf blocks (BLEVEL)
- Number of leaf blocks (LEAF_BLOCKS)
- Number of distinct index values (DISTINCT_KEYS)
- Average number of leaf blocks for each index value (AVG_LEAF_BLOCKS_PER_KEY)
- Average number of data blocks for each index value (for an index on a table) (AVG_DATA_BLOCKS_PER_KEY)
- Clustering factor (how well ordered the rows are about the indexed values) (CLUSTERING_FACTOR)

For domain indexes, this statement invokes the user-defined statistics collection function specified in the statistics type associated with the index (see ASSOCIATE STATISTICS on page 12-48). If no statistics type is associated with the domain index, then the statistics type associated with its indextype is used. If no statistics type exists for either the index or its indextype, then no user-defined statistics are collected. User-defined index statistics appear in the STATISTICS column of the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

Restriction on analyzing an index: You cannot analyze a domain index that is marked IN_PROGRESS or FAILED.

See Also:

- CREATE INDEX on page 13-62 for more information on domain indexes
- *Oracle9i Database Reference* for information on the data dictionary views
- "Analyzing an Index: Example" on page 12-46

TABLE *table*

Specify a table to be analyzed. When you collect statistics for a table, Oracle also automatically collects the statistics for each of the table's indexes and domain indexes, as long as no *for_clauses* are used.

When you analyze a table, Oracle collects statistics about expressions occurring in any function-based indexes as well. Therefore, be sure to create function-based indexes on the table before analyzing the table.

See Also: CREATE INDEX on page 13-62 for more information about function-based indexes

When analyzing a table, Oracle skips all domain indexes marked LOADING or FAILED.

For an index-organized table, Oracle also analyzes any mapping table and calculates its PCT_ACCESSS_DIRECT statistics. These statistics estimate the accuracy of "guess" data block addresses stored as part of the local rowids in the mapping table.

Oracle collects the following statistics for a table. Statistics marked with an asterisk are always computed exactly. Table statistics, including the status of domain indexes, appear in the data dictionary views USER_TABLES, ALL_TABLES, and DBA_TABLES in the columns shown in parentheses.

- Number of rows (NUM_ROWS)
- * Number of data blocks below the high water mark (that is, the number of data blocks that have been formatted to receive data, regardless whether they currently contain data or are empty) (BLOCKS)
- * Number of data blocks allocated to the table that have never been used (EMPTY_BLOCKS)
- Average available free space in each data block in bytes (AVG_SPACE)
- Number of chained rows (CHAIN_COUNT)
- Average row length, including the row's overhead, in bytes (AVG_ROW_LEN)

Restrictions on analyzing a table:

- You cannot use ANALYZE to collect statistics on data dictionary tables.
- You cannot use ANALYZE to collect statistics on an external table. However, you can use the DBMS_STATS package for this purpose.
- You cannot use ANALYZE to collect default statistics on a temporary table. However, if you have created an association between one or more columns of a temporary table and a user-defined statistics type, then you can use ANALYZE to collect the user-defined statistics on the temporary table. (The association must already exist.)
- You cannot compute or estimate statistics for the following column types: REFs, varrays, nested tables, LOBs (LOBs are not analyzed, they are skipped), LONGs, or object types. However, if a statistics type is associated with such a column, then user-defined statistics are collected.

See Also:

- ASSOCIATE STATISTICS on page 12-48
- *Oracle9i Database Reference* for information on the data dictionary views

PARTITION | SUBPARTITION

Specify the *partition* or *subpartition* on which you want statistics to be gathered. You cannot use this clause when analyzing clusters.

If you specify PARTITION and *table* is composite-partitioned, then Oracle analyzes all the subpartitions within the specified partition.

CLUSTER *cluster*

Specify a cluster to be analyzed. When you collect statistics for a cluster, Oracle also automatically collects the statistics for all the cluster's tables and all their indexes, including the cluster index.

For both indexed and hash clusters, Oracle collects the average number of data blocks taken up by a single cluster key (AVG_BLOCKS_PER_KEY). These statistics appear in the data dictionary views ALL_CLUSTERS, USER_CLUSTERS and DBA_CLUSTERS.

See Also: *Oracle9i Database Reference* for information on the data dictionary views and "Analyzing a Cluster: Example" on page 12-47

compute_statistics_clause

COMPUTE STATISTICS instructs Oracle to compute exact statistics about the analyzed object and store them in the data dictionary. When you analyze a table, both table and column statistics are collected.

Both computed and estimated statistics are used by the Oracle optimizer to choose the execution plan for SQL statements that access analyzed objects. These statistics may also be useful to application developers who write such statements.

Specify SYSTEM if you want Oracle to compute only system (not user-defined statistics). If you omit SYSTEM, then Oracle collects both system-generated statistics and statistics generated by the collection functions declared in a statistics type.

See Also:

- *Oracle9i Data Cartridge Developer's Guide* for information on creating statistics collection functions
- *Oracle9i Database Performance Tuning Guide and Reference* for information on how these statistics are used
- "Computing Statistics: Examples" on page 12-45

for_clause

The *for_clause* lets you specify whether an entire table or index, or just particular columns, will be analyzed. The following clauses apply only to the ANALYZE TABLE version of this statement.

FOR TABLE Specify FOR TABLE to restrict the statistics collected to only table statistics rather than table and column statistics.

FOR COLUMNS Specify FOR COLUMNS to restrict the statistics collected to only column statistics for the specified columns and scalar object attributes, rather than for all columns and attributes; *attribute* specifies the qualified column name of an item in an object.

FOR ALL COLUMNS Specify FOR ALL COLUMNS to collect column statistics for all columns and scalar object attributes.

FOR ALL INDEXED COLUMNS Specify FOR ALL INDEXED COLUMNS to collect column statistics for all indexed columns in the table.

Column statistics can be based on the entire column or can use a histogram by specifying SIZE.

Oracle collects the following column statistics:

- Number of distinct values in the column as a whole
- Maximum and minimum values in each band

See Also: *Oracle9i Database Performance Tuning Guide and Reference* and "Creating Histograms: Examples" on page 12-46 for more information on histograms

Column statistics appear in the data dictionary views USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS. Histograms appear in the data dictionary

views USER_TAB_HISTOGRAMS, DBA_TAB_HISTOGRAMS, and ALL_TAB_HISTOGRAMS; USER_PART_HISTOGRAMS, DBA_PART_HISTOGRAMS, and ALL_PART_HISTOGRAMS; and USER_SUBPART_HISTOGRAMS, DBA_SUBPART_HISTOGRAMS, and ALL_SUBPART_HISTOGRAMS.

> **Note:** MAXVALUE and MINVALUE columns of USER_, DBA_, and ALL_TAB_COLUMNS have a length of 32 bytes. If you analyze columns with a length >32 bytes, and if the columns are padded with leading blanks, then Oracle may take into account only the leading blanks and return unexpected statistics.

If a user-defined statistics type has been associated with any columns, then the *for_clause* collects user-defined statistics using that statistics type. If no statistics type is associated with a column, then Oracle checks to see if any statistics type has been associated with the type of the column, and uses that statistics type. If no statistics type has been associated with either the column or its user-defined type, then no user-defined statistics are collected. User-defined column statistics appear in the STATISTICS column of the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

If you want to collect statistics on both the table as a whole and on one or more columns, then be sure to generate the statistics for the table first, and then for the columns. Otherwise, the table-only ANALYZE will overwrite the histograms generated by the column ANALYZE. For example, issue the following statements:

```
ANALYZE TABLE emp ESTIMATE STATISTICS;
ANALYZE TABLE emp ESTIMATE STATISTICS
   FOR ALL COLUMNS;
```

FOR ALL INDEXES Specify FOR ALL INDEXES if you want all indexes associated with the table to be analyzed.

FOR ALL LOCAL INDEXES Specify FOR ALL LOCAL INDEXES if you want all local index partitions to be analyzed. You must specify the keyword LOCAL if the PARTITION clause and INDEX are specified.

SIZE Specify the maximum number of buckets in the histogram. The default value is 75, minimum value is 1, and maximum value is 254.

Note: Oracle does not create a histogram with more buckets than the number of rows in the sample. Also, if the sample contains any values that are very repetitious, then Oracle creates the specified number of buckets, but the value indicated by the NUM_BUCKETS column of the ALL_, DBA_, and USER_TAB_COLUMNS views may be smaller because of an internal compression algorithm.

See Also: "Creating Histograms: Examples" on page 12-46

estimate_statistics_clause

ESTIMATE STATISTICS instructs Oracle to estimate statistics about the analyzed object and store them in the data dictionary.

Both computed and estimated statistics are used by the Oracle optimizer to choose the execution plan for SQL statements that access analyzed objects. These statistics may also be useful to application developers who write such statements.

Specify SYSTEM if you want Oracle to estimate only system (not user-defined statistics). If you omit SYSTEM, then Oracle estimates both system-generated statistics and statistics generated by the collection functions declared in a statistics type.

See Also: *Oracle9i Data Cartridge Developer's Guide* for information on creating statistics collection functions and "Estimating Statistics: Example" on page 12-46

for_clause See the description under *compute_statistics_clause* on page 12-38

SAMPLE Specify the amount of data from the analyzed object Oracle should sample to estimate statistics. If you omit this parameter, then Oracle samples 1064 rows.

The default sample value is adequate for tables up to a few thousand rows. If your tables are larger, specify a higher value for SAMPLE. If you specify more than half of the data, then Oracle reads all the data and computes the statistics.

- ROWS causes Oracle to sample *integer* rows of the table or cluster or *integer* entries from the index. The integer must be at least 1.

- PERCENT causes Oracle to sample *integer* percent of the rows from the table or cluster or *integer* percent of the index entries. The integer can range from 1 to 99.

  See Also: *Oracle9i Database Performance Tuning Guide and Reference* for information on how these statistics are used

validation_clauses

The validation clauses let you validate REFs and the structure of the analyzed object.

VALIDATE REF UPDATE Clause

Specify VALIDATE REF UPDATE to validate the REFs in the specified table, check the rowid portion in each REF, compare it with the true rowid, and correct, if necessary. You can use this clause only when analyzing a table.

SET DANGLING TO NULL SET DANGLING TO NULL sets to NULL any REFs (whether or not scoped) in the specified table that are found to point to an invalid or nonexistent object.

Note: If the owner of the table does not have SELECT object privilege on the referenced objects, then Oracle will consider them invalid and set them to NULL. Subsequently these REFs will not be available in a query, even if it is issued by a user with appropriate privileges on the objects.

VALIDATE STRUCTURE

Specify VALIDATE STRUCTURE to validate the structure of the analyzed object. The statistics collected by this clause are not used by the Oracle optimizer, as are statistics collected by the COMPUTE STATISTICS and ESTIMATE STATISTICS clauses.

See Also: "Validating a Table: Example" on page 12-46

- For a table, Oracle verifies the integrity of each of the table's data blocks and rows. For an index-organized table, Oracle also generates compression statistics (optimal prefix compression count) for the primary key index on the table.
- For a cluster, Oracle automatically validates the structure of the cluster's tables.

- For a partitioned table, Oracle also verifies that each row belongs to the correct partition. If a row does not collate correctly, then its rowid is inserted into the INVALID_ROWS table.
- For a temporary table, Oracle validates the structure of the table and its indexes during the current session.
- For an index, Oracle verifies the integrity of each data block in the index and checks for block corruption. This clause does not confirm that each row in the table has an index entry or that each index entry points to a row in the table. You can perform these operations by validating the structure of the table with the CASCADE clause.

  Oracle also computes compression statistics (optimal prefix compression count) for all normal indexes

  Oracle stores statistics about the index in the data dictionary views INDEX_STATS and INDEX_HISTOGRAM.

  > **See Also:** *Oracle9i Database Reference* for information on these views

If Oracle encounters corruption in the structure of the object, then an error message is returned to you. In this case, drop and re-create the object.

INTO The INTO clause of VALIDATE STRUCTURE is valid only for partitioned tables. Specify a table into which Oracle lists the rowids of the partitions whose rows do not collate correctly. If you omit *schema*, then Oracle assumes the list is in your own schema. If you omit this clause altogether, then Oracle assumes that the table is named INVALID_ROWS. The SQL script used to create this table is UTLVALID.SQL.

CASCADE Specify CASCADE if you want Oracle to validate the structure of the indexes associated with the table or cluster. If you use this clause when validating a table, then Oracle also validates the table's indexes. If you use this clause when validating a cluster, then Oracle also validates all the clustered tables' indexes, including the cluster index.

If you use this clause to validate an enabled (but previously disabled) function-based index, then validation errors may result. In this case, you must rebuild the index.

ONLINE | OFFLINE Specify ONLINE to enable Oracle to run the validation while DML operations are ongoing within the object. Oracle reduces the amount of validation performed to allow for concurrency.

Specify OFFLINE, to maximize the amount of validation performed. This setting prevents INSERT, UPDATE, and DELETE statements from concurrently accessing the object during validation but allows queries. This is the default.

Restriction on ONLINE: You cannot specify ONLINE when analyzing a clustered object.

LIST CHAINED ROWS

LIST CHAINED ROWS lets you identify migrated and chained rows of the analyzed table or cluster. You cannot use this clause when analyzing an index.

In the INTO clause, specify a table into which Oracle lists the migrated and chained rows. If you omit *schema*, then Oracle assumes the list table is in your own schema. If you omit this clause altogether, then Oracle assumes that the table is named CHAINED_ROWS. The list table must be on your local database.

You can create the CHAINED_ROWS table using one of these scripts:

- UTLCHAIN.SQL uses physical rowids. Therefore it can accommodate rows from conventional tables but not from index-organized tables. (See the Note that follows.)
- UTLCHN1.SQL uses universal rowids, so it can accommodate rows from both conventional and index-organized tables.

If you create your own chained-rows table, then it must follow the format prescribed by one of these two scripts.

Note: If you are analyzing index-organized tables based on primary keys (rather than universal rowids), then you must create a separate chained-rows table for each index-organized table to accommodate its primary-key storage. Use the SQL scripts DBMSIOTC.SQL and PRVTIOTC.PLB to define the BUILD_CHAIN_ROWS_TABLE procedure, and then execute this procedure to create an IOT_CHAINED_ROWS table for each such index-organized table.

See Also:

- *Oracle9i Database Migration* for compatibility issues related to the use of the UTL* scripts
- The DBMS_IOT package in *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the packaged SQL scripts
- *Oracle9i Database Administrator's Guide* for information on eliminating migrated and chained rows
- "Listing Chained Rows: Example" on page 12-47

DELETE STATISTICS

Specify DELETE STATISTICS to delete any statistics about the analyzed object that are currently stored in the data dictionary. Use this statement when you no longer want Oracle to use the statistics.

When you use this clause on a table, Oracle also automatically removes statistics for all the table's indexes. When you use this clause on a cluster, Oracle also automatically removes statistics for all the cluster's tables and all their indexes, including the cluster index.

Specify SYSTEM if you want Oracle to delete only system (not user-defined statistics). If you omit SYSTEM, and if user-defined column or index statistics were collected for an object, then Oracle also removes the user-defined statistics by invoking the statistics deletion function specified in the statistics type that was used to collect the statistics.

See Also: "Deleting Statistics: Example" on page 12-46

Examples

Computing Statistics: Examples The following statement computes statistics for the sample table oe.orders:

```
ANALYZE TABLE orders COMPUTE STATISTICS;
```

The following statement computes only system statistics on the sample table oe.orders:

```
ANALYZE TABLE orders COMPUTE SYSTEM STATISTICS;
```

The following statement calculates statistics for a scalar object attribute: ANALYZE TABLE customers COMPUTE STATISTICS

```
   FOR COLUMNS cust_address.postal_code;
```

Estimating Statistics: Example The following statement estimates statistics for the sample table `oe.orders` and all of its indexes:

```
ANALYZE TABLE orders ESTIMATE STATISTICS;
```

Deleting Statistics: Example The following statement deletes statistics about the sample table `oe.orders` and all its indexes from the data dictionary:

```
ANALYZE TABLE orders DELETE STATISTICS;
```

Creating Histograms: Examples The following statement creates a 10-band histogram on the `location_id` column of the sample table `hr.locations`:

```
ANALYZE TABLE locations
   COMPUTE STATISTICS FOR COLUMNS country_id SIZE 10;
```

You can then query the USER_TAB_COLUMNS data dictionary view to retrieve statistics:

```
SELECT NUM_DISTINCT, NUM_BUCKETS, SAMPLE_SIZE
   FROM USER_TAB_COLUMNS
   WHERE TABLE_NAME = 'LOCATIONS' AND COLUMN_NAME = 'COUNTRY_ID';

NUM_DISTINCT NUM_BUCKETS SAMPLE_SIZE
------------ ----------- -----------
          14           7          23
```

Depending on the size of your table, even though the ANALYZE statement specified 10 buckets, Oracle may create fewer buckets that you specify in the ANALYZE statement. For an explanation, see the note on SIZE on page 12-40.

You can also collect histograms for a single partition of a table. The following statement analyzes partition `sales_q2_2000` of the sample table `sh.sales`:

```
ANALYZE TABLE sales PARTITION (sales_q2_2000) COMPUTE STATISTICS;
```

Analyzing an Index: Example The following statement validates the structure of the sample index `oe.inv_product_ix`:

```
ANALYZE INDEX inv_product_ix VALIDATE STRUCTURE;
```

Validating a Table: Example The following statement analyzes the sample table `hr.employees` and all of its indexes:

```
ANALYZE TABLE employees VALIDATE STRUCTURE CASCADE;
```

For a table, the VALIDATE REF UPDATE clause verifies the REFs in the specified table, checks the rowid portion of each REF, and then compares it with the true rowid. If the result is an incorrect rowid, then the REF is updated so that the rowid portion is correct.

The following statement validates the REFs in the sample table `oe.customers`:

```
ANALYZE TABLE customers VALIDATE REF UPDATE;
```

The following statements validates the structure of the sample table `oe.customers` while allowing simultaneous DML:

```
ANALYZE TABLE customers VALIDATE STRUCTURE ONLINE;
```

Analyzing a Cluster: Example The following statement analyzes the `personnel` cluster (created in "Creating a Cluster: Example" on page 13-9), all of its tables, and all of their indexes, including the cluster index:

```
ANALYZE CLUSTER personnel
    VALIDATE STRUCTURE CASCADE;
```

Listing Chained Rows: Example The following statement collects information about all the chained rows of the table `orders`:

```
ANALYZE TABLE orders
   LIST CHAINED ROWS INTO chained_rows;
```

The preceding statement places the information into the table `chained_rows`. You can then examine the rows with this query (no rows will be returned if the table contains no chained rows):

```
SELECT owner_name, table_name, head_rowid, analyze_timestamp
    FROM chained_rows;

OWNER_NAME  TABLE_NAME  HEAD_ROWID         ANALYZE_TIMESTAMP
----------  ----------  ------------------ -----------------
OE          ORDERS      AAAAZzAABAAABrXAAA 25-SEP-2000
```

ASSOCIATE STATISTICS

Purpose

Use the ASSOCIATE STATISTICS statement to associate a statistics type (or default statistics) containing functions relevant to statistics collection, selectivity, or cost with one or more columns, standalone functions, packages, types, domain indexes, or indextypes.

For a listing of all current statistics type associations, query the USER_ASSOCIATIONS data dictionary view. If you analyze the object with which you are associating statistics, then you can also query the associations in the USER_USTATS view.

> **See Also:** ANALYZE on page 12-33 for information on the order of precedence with which ANALYZE uses associations

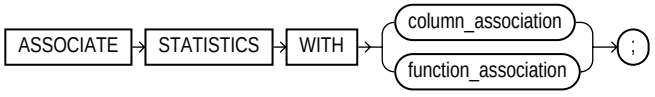
Prerequisites

To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype). In addition, unless you are associating only default statistics, you must have execute privilege on the statistics type. The statistics type must already have been defined.

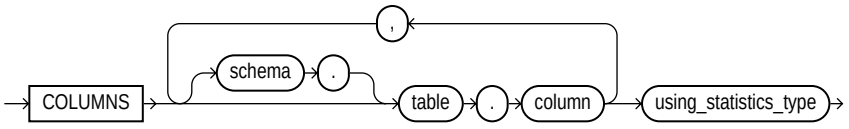
> **See Also:** CREATE TYPE on page 16-3 for information on defining types

Syntax

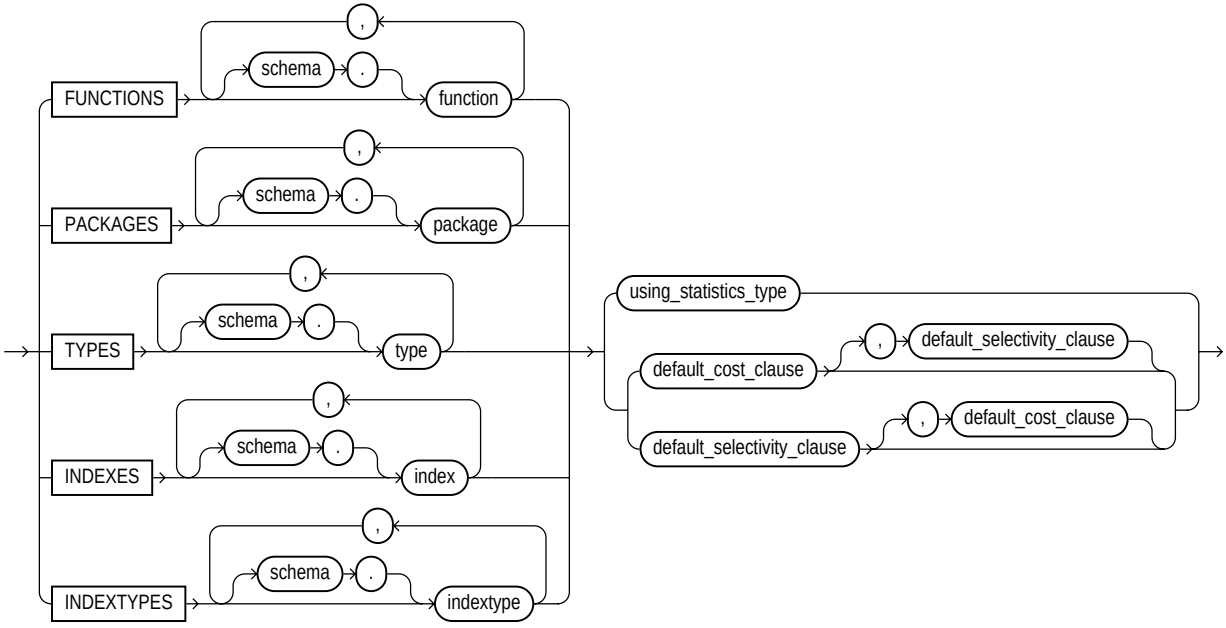
associate_statistics::=



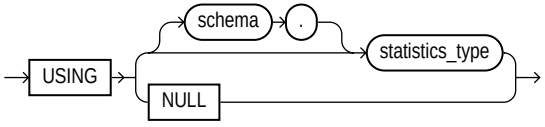
column_association::=



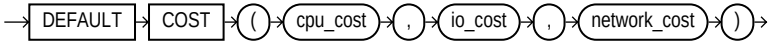
function_association::=



using_statistics_type::=



default_cost_clause::=



default_selectivity_clause::=

Keywords and Parameters

column_association

Specify one or more table columns. If you do not specify *schema*, then Oracle assumes the table is in your own schema.

function_association

Specify one or more standalone functions, packages, user-defined datatypes, domain indexes, or indextypes. If you do not specify *schema*, then Oracle assumes the object is in your own schema.

- FUNCTIONS refers only to standalone functions, not to method types or to built-in functions.
- TYPES refers only to user-defined types, not to built-in SQL datatypes.

Restriction on *function_association*: You cannot specify an object for which you have already defined an association. You must first disassociate the statistics from this object.

> **See Also:** DISASSOCIATE STATISTICS on page 16-64
> "Associating Statistics: Example" on page 12-51

using_statistics_type

Specify the statistics type (or a synonym for the type) being associated with column, function, package, type, domain index, or indextype. The *statistics_type* must already have been created.

The NULL keyword is valid only when you are associating statistics with a column or an index. When you associate a statistics type with an object type, columns of that object type inherit the statistics type. Likewise, when you associate a statistics type with an indextype, index instances of the indextype inherit the statistics type.You can override this inheritance by associating a different statistics type for the column or index. Alternatively, if you do not want to associate any statistics type for the column or index, then you can specify NULL in the *using_statistics_type* clause.

Restriction on the *using_statistics_type* **clause:** You cannot specify NULL for functions, packages, types, or indextypes.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for information on creating statistics collection functions

default_cost_clause

Specify default costs for standalone functions, packages, types, domain indexes, or indextypes. If you specify this clause, then you must include one number each for CPU cost, I/O cost, and network cost, in that order. Each cost is for a single execution of the function or method or for a single domain index access. Accepted values are integers of zero or greater.

default_selectivity_clause

Specify as a percent the default selectivity for predicates with standalone functions, types, packages, or user-defined operators. The *default_selectivity* must be a number between 0 and 100. Values outside this range are ignored.

Restriction on the *default_selectivity_clause***:** You cannot specify DEFAULT SELECTIVITY for domain indexes or indextypes.

See Also: "Specifying Default Cost: Example" on page 12-51

Examples

Associating Statistics: Example This statement creates an association for the standalone package emp_mgmt (created in "Creating a Package: Example" on page 14-53):

```
ASSOCIATE STATISTICS WITH PACKAGES emp_mgmt DEFAULT SELECTIVITY 10;
```

Specifying Default Cost: Example This statement specifies that using the domain index t_a to implement a given predicate always has a CPU cost of 100, I/O of 5, and network cost of 0.

```
ASSOCIATE STATISTICS WITH INDEXES t_a DEFAULT COST (100,5,0);
```

The optimizer will simply use these default costs instead of calling a cost function.

AUDIT

Purpose

Use the AUDIT statement to:

- Track the occurrence of SQL statements in subsequent user sessions. You can track the occurrence of a specific SQL statement or of all SQL statements authorized by a particular system privilege. Auditing operations on SQL statements apply only to subsequent sessions, not to current sessions.
- Track operations on a specific schema object. Auditing operations on schema objects apply to current sessions as well as to subsequent sessions.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the DBMS_FGA package, which lets you create and administer value-based auditing policies
- NOAUDIT on page 17-82 for information on disabling auditing of SQL statement

Prerequisites

To audit occurrences of a SQL statement, you must have AUDIT SYSTEM system privilege.

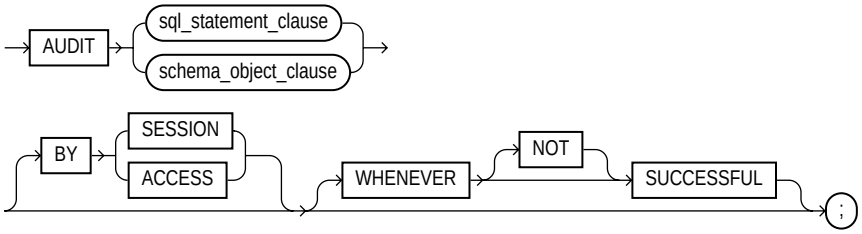
To audit operations on a schema object, the object you choose for auditing must be in your own schema or you must have AUDIT ANY system privilege. In addition, if the object you choose for auditing is a directory object, even if you created it, then you must have AUDIT ANY system privilege.

To collect auditing results, you must set the initialization parameter AUDIT_TRAIL to DB. You can specify auditing options regardless of whether auditing is enabled. However, Oracle does not generate audit records until you enable auditing.

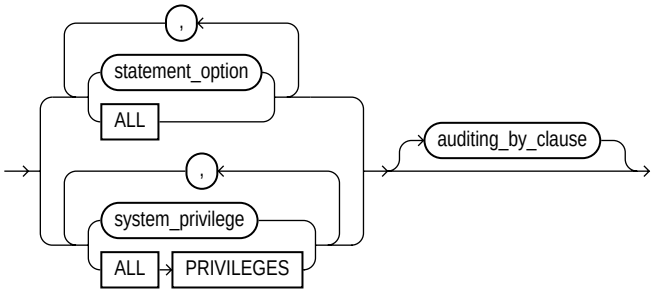
See Also: *Oracle9i Database Reference* for information on the AUDIT_TRAIL parameter

Syntax

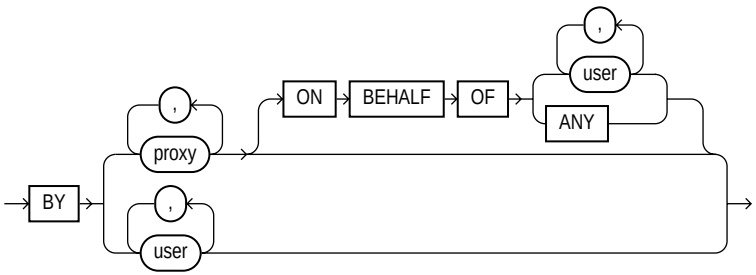
audit::=



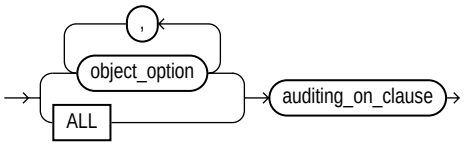
sql_statement_clause::=



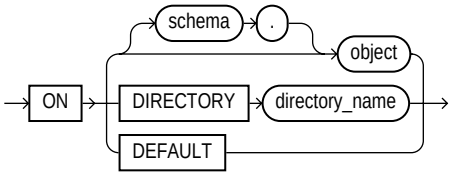
auditing_by_clause::=



schema_object_clause::=



auditing_on_clause::=



Keywords and Parameters

sql_statement_clause

Use the *sql_statement_clause* to audit SQL statements.

statement_option

Specify a statement option to audit specific SQL statements.

For each audited operation, Oracle produces an audit record containing this information:

- The user performing the operation
- The type of operation
- The object involved in the operation
- The date and time of the operation

Oracle writes audit records to the audit trail, which is a database table containing audit records. You can review database activity by examining the audit trail through data dictionary views.

See Also:

- Table 12–1 on page 12-58 and Table 12–2 on page 12-60 for a list of statement options and the SQL statements they audit
- *Oracle9i Database Administrator's Guide*for a listing of the audit trail data dictionary views
- *Oracle9i Database Reference* for detailed descriptions of the data dictionary views
- "Auditing SQL Statements Relating to Roles: Example" on page 12-62

system_privilege

Specify a system privilege to audit SQL statements that are authorized by the specified system privilege.

Rather than specifying many individual system privileges, you can specify the roles CONNECT, RESOURCE, and DBA. Doing so is equivalent to auditing all of the system privileges granted to those roles.

Oracle also provides two shortcuts for specifying groups of system privileges and statement options at once:

ALL Specify ALL to audit all statements options shown in Table 12–1 but not the additional statement options shown in Table 12–2.

ALL PRIVILEGES Specify ALL PRIVILEGES to audit system privileges.

Note: Oracle Corporation recommends that you specify individual system privileges and statement options for auditing rather than roles or shortcuts. The specific system privileges and statement options encompassed by roles and shortcuts change from one release to the next and may not be supported in future versions of Oracle.

See Also:

- Table 17–1, " System Privileges" for a list of all system privileges and the SQL statements that they authorize
- GRANT on page 17-29 for more information on the CONNECT, RESOURCE, and DBA roles
- "Auditing Query and Update SQL Statements: Example" on page 12-63, "Auditing Deletions: Example" on page 12-63, "Auditing Statements Relating to Directories: Examples" on page 12-63

auditing_by_clause

Specify the *auditing_by_clause* to audit only those SQL statements issued by particular users. If you omit this clause, then Oracle audits all users' statements.

BY user Use this clause to restrict auditing to only SQL statements issued by the specified users.

BY proxy Use this clause to restrict auditing to only SQL statements issued by the specified proxies.

> **See Also:** *Oracle9i Database Concepts* for more information on proxies and their use of the database

ON BEHALF OF Specify *user* to indicate auditing of statements executed on behalf of a particular user. ANY indicates auditing of statements executed on behalf of any user.

schema_object_clause

Use the *schema_object_clause* to audit operations on schema objects.

object_option

Specify the particular operation for auditing. Table 12–3 on page 12-62 shows each object option and the types of objects to which it applies. The name of each object option specifies a SQL statement to be audited. For example, if you choose to audit a table with the ALTER option, then Oracle audits all ALTER TABLE statements issued against the table. If you choose to audit a sequence with the SELECT option, then Oracle audits all statements that use any of the sequence's values.

ALL

Specify ALL as a shortcut equivalent to specifying all object options applicable for the type of object.

auditing_on_clause

The *auditing_on_clause* lets you specify the particular schema object to be audited.

> **See Also:** "Auditing Queries on a Table: Example" on page 12-63, "Auditing Inserts and Updates on a Table: Example" on page 12-64, and "Auditing Operations on a Sequence: Example" on page 12-64

schema Specify the schema containing the object chosen for auditing. If you omit *schema*, then Oracle assumes the object is in your own schema.

object Specify the name of the object to be audited. The object must be a table, view, sequence, stored procedure, function, package, materialized view, or library.

You can also specify a synonym for a table, view, sequence, procedure, stored function, package materialized view, or user-defined type.

ON DEFAULT Specify ON DEFAULT to establish the specified object options as default object options for subsequently created objects. Once you have established these default auditing options, any subsequently created object is automatically audited with those options. The default auditing options for a view are always the union of the auditing options for the view's base tables. You can see the current default auditing options by querying the ALL_DEF_AUDIT_OPTS data dictionary view.

If you change the default auditing options, then the auditing options for previously created objects remain the same. You can change the auditing options for an existing object only by specifying the object in the ON clause of the AUDIT statement.

> **See Also:** "Setting Default Auditing Options: Example" on page 12-64

ON DIRECTORY *directory_name* The ON DIRECTORY clause lets you specify the name of a directory chosen for auditing.

BY SESSION

Specify BY SESSION if you want Oracle to write a single record for all SQL statements of the same type issued and operations of the same type executed on the same schema objects in the same session.

> **Note:** Oracle can write to an operating system audit file but cannot read it to detect whether an entry has already been written for a particular operation. Therefore, if you are using an operating system file for the audit trail (that is, the AUDIT_FILE_DEST initialization parameter is set to OS), then Oracle may write multiple records to the audit trail file even if you specify BY SESSION.

BY ACCESS

Specify BY ACCESS if you want Oracle to write one record for each audited statement and operation.

If you specify statement options or system privileges that audit data definition language (DDL) statements, then Oracle automatically audits by access regardless of whether you specify the BY SESSION clause or BY ACCESS clause.

For statement options and system privileges that audit SQL statements other than DDL, you can specify either BY SESSION or BY ACCESS. BY SESSION is the default.

WHENEVER [NOT] SUCCESSFUL

Specify WHENEVER SUCCESSFUL to audit only SQL statements and operations that succeed.

Specify WHENEVER NOT SUCCESSFUL to audit only statements and operations that fail or result in errors.

If you omit this clause, then Oracle performs the audit regardless of success or failure.

Tables of Auditing Options

Table 12–1 Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
CLUSTER	CREATE CLUSTER AUDIT CLUSTER DROP CLUSTER TRUNCATE CLUSTER
CONTEXT	CREATE CONTEXT DROP CONTEXT
DATABASE LINK	CREATE DATABASE LINK DROP DATABASE LINK
DIMENSION	CREATE DIMENSION ALTER DIMENSION DROP DIMENSION
DIRECTORY	CREATE DIRECTORY DROP DIRECTORY
INDEX	CREATE INDEX ALTER INDEX DROP INDEX

Table 12–1 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
NOT EXISTS	All SQL statements that fail because a specified object does not exist.
PROCEDURE[a]	CREATE FUNCTION CREATE LIBRARY CREATE PACKAGE CREATE PACKAGE BODY CREATE PROCEDURE DROP FUNCTION DROP LIBRARY DROP PACKAGE DROP PROCEDURE
PROFILE	CREATE PROFILE ALTER PROFILE DROP PROFILE
PUBLIC DATABASE LINK	CREATE PUBLIC DATABASE LINK DROP PUBLIC DATABASE LINK
PUBLIC SYNONYM	CREATE PUBLIC SYNONYM DROP PUBLIC SYNONYM
ROLE	CREATE ROLE ALTER ROLE DROP ROLE SET ROLE
ROLLBACK SEGMENT	CREATE ROLLBACK SEGMENT ALTER ROLLBACK SEGMENT DROP ROLLBACK SEGMENT
SEQUENCE	CREATE SEQUENCE DROP SEQUENCE
SESSION	Logons
SYNONYM	CREATE SYNONYM DROP SYNONYM
SYSTEM AUDIT	AUDIT *sql_statements* NOAUDIT *sql_statements*

Table 12–1 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
SYSTEM GRANT	GRANT *system_privileges_and_roles* REVOKE *system_privileges_and_roles*
TABLE	CREATE TABLE DROP TABLE TRUNCATE TABLE
TABLESPACE	CREATE TABLESPACE ALTER TABLESPACE DROP TABLESPACE
TRIGGER	CREATE TRIGGER ALTER TRIGGER with ENABLE and DISABLE clauses DROP TRIGGER ALTER TABLE with ENABLE ALL TRIGGERS clause and DISABLE ALL TRIGGERS clause
TYPE	CREATE TYPE CREATE TYPE BODY ALTER TYPE DROP TYPE DROP TYPE BODY
USER	CREATE USER ALTER USER DROP USER
VIEW	CREATE VIEW DROP VIEW

[a]Java schema objects (sources, classes, and resources) are considered the same as procedures for purposes of auditing SQL statements.

Table 12–2 Additional Statement Auditing Options for SQL Statements

Statement Option	SQL Statements and Operations
ALTER SEQUENCE	ALTER SEQUENCE

Table 12–2 (Cont.) Additional Statement Auditing Options for SQL Statements

Statement Option	SQL Statements and Operations
ALTER TABLE	ALTER TABLE
COMMENT TABLE	COMMENT ON TABLE *table*, *view*, *materialized view* COMMENT ON COLUMN *table.column*, *view.column*, *materialized view.column*
DELETE TABLE	DELETE FROM *table*, *view*
EXECUTE PROCEDURE	CALL Execution of any procedure or function or access to any variable, library, or cursor inside a package.
GRANT DIRECTORY	GRANT privilege ON directory REVOKE privilege ON directory
GRANT PROCEDURE	GRANT privilege ON procedure, function, package REVOKE privilege ON procedure, function, package
GRANT SEQUENCE	GRANT privilege ON sequence REVOKE privilege ON sequence
GRANT TABLE	GRANT privilege ON table, view, materialized view. REVOKE privilege ON table, view, materialized view
GRANT TYPE	GRANT privilege ON TYPE REVOKE privilege ON TYPE
INSERT TABLE	INSERT INTO table, view
LOCK TABLE	LOCK TABLE table, view
SELECT SEQUENCE	Any statement containing sequence.CURRVAL or sequence.NEXTVAL
SELECT TABLE	SELECT FROM table, view, materialized view
UPDATE TABLE	UPDATE table, view

Table 12–3 Object Auditing Options

Object Option	Table	View	Sequence	Procedure, Function, Package[a]	Material-ized View	Directory	Library	Object Type	Context
ALTER	X	—	X	—	X	—	—	X	—
AUDIT	X	X	X	X	X	X	—	X	X
COMMENT	X	X	—	—	X	—	—	—	—
DELETE	X	X	—	—	X	—	—	—	—
EXECUTE	—	—	—	X	—	—	X	—	—
GRANT	X	X	X	X	—	X	X	X	X
INDEX	X	—	—	—	X	—	—	—	—
INSERT	X	X	—	—	X	—	—	—	—
LOCK	X	X	—	—	X	—	—	—	—
READ	—	—	—	—	—	X	—	—	—
RENAME	X	X	—	X	X	—	—	—	—
SELECT	X	X	X	—	X	—	—	—	—
UPDATE	X	X	—	—	X	—	—	—	—

[a] Java schema objects (sources, classes, and resources) are considered the same as procedures, functions, and packages for purposes of auditing options.

Examples

Auditing SQL Statements Relating to Roles: Example To choose auditing for every SQL statement that creates, alters, drops, or sets a role, regardless of whether the statement completes successfully, issue the following statement:

```
AUDIT ROLE;
```

To choose auditing for every statement that successfully creates, alters, drops, or sets a role, issue the following statement:

```
AUDIT ROLE
    WHENEVER SUCCESSFUL;
```

To choose auditing for every CREATE ROLE, ALTER ROLE, DROP ROLE, or SET ROLE statement that results in an Oracle error, issue the following statement:

```
AUDIT ROLE
    WHENEVER NOT SUCCESSFUL;
```

Auditing Query and Update SQL Statements: Example To choose auditing for any statement that queries or updates any table, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE;
```

To choose auditing for statements issued by the users hr and oe that query or update a table or view, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE
    BY hr, oe;
```

Auditing Deletions: Example To choose auditing for statements issued using the DELETE ANY TABLE system privilege, issue the following statement:

```
AUDIT DELETE ANY TABLE;
```

Auditing Statements Relating to Directories: Examples To choose auditing for statements issued using the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT CREATE ANY DIRECTORY;
```

To choose auditing for CREATE DIRECTORY (and DROP DIRECTORY) statements that do not use the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT DIRECTORY;
```

To choose auditing for every statement that reads files from the bfile_dir directory, issue the following statement:

```
AUDIT READ ON DIRECTORY bfile_dir;
```

Auditing Queries on a Table: Example To choose auditing for every SQL statement that queries the employees table in the schema hr, issue the following statement:

```
AUDIT SELECT
    ON hr.employees;
```

To choose auditing for every statement that successfully queries the employees table in the schema hr, issue the following statement:

```
AUDIT SELECT
    ON hr.employees
    WHENEVER SUCCESSFUL;
```

To choose auditing for every statement that queries the employees table in the schema hr and results in an Oracle error, issue the following statement:

```
AUDIT SELECT
    ON hr.employees
    WHENEVER NOT SUCCESSFUL;
```

Auditing Inserts and Updates on a Table: Example To choose auditing for every statement that inserts or updates a row in the customers table in the schema oe, issue the following statement:

```
AUDIT INSERT, UPDATE
    ON oe.customers;
```

Auditing Operations on a Sequence: Example To choose auditing for every statement that performs any operation on the employees_seq sequence in the schema hr, issue the following statement:

```
AUDIT ALL
    ON hr.employees_seq;
```

The preceding statement uses the ALL shortcut to choose auditing for the following statements that operate on the sequence:

- ALTER SEQUENCE
- AUDIT
- GRANT
- any statement that accesses the sequence's values using the pseudocolumns CURRVAL or NEXTVAL

Setting Default Auditing Options: Example The following statement specifies default auditing options for objects created in the future:

```
AUDIT ALTER, GRANT, INSERT, UPDATE, DELETE
    ON DEFAULT;
```

Any objects created later are automatically audited with the specified options that apply to them, if auditing has been enabled:

- If you create a table, then Oracle automatically audits any ALTER, GRANT, INSERT, UPDATE, or DELETE statements issued against the table.
- If you create a view, then Oracle automatically audits any GRANT, INSERT, UPDATE, or DELETE statements issued against the view.
- If you create a sequence, then Oracle automatically audits any ALTER or GRANT statements issued against the sequence.
- If you create a procedure, package, or function, then Oracle automatically audits any ALTER or GRANT statements issued against it.

CALL

Purpose

Use the CALL statement to execute a **routine** (a standalone procedure or function, or a procedure or function defined within a type or package) from within SQL.

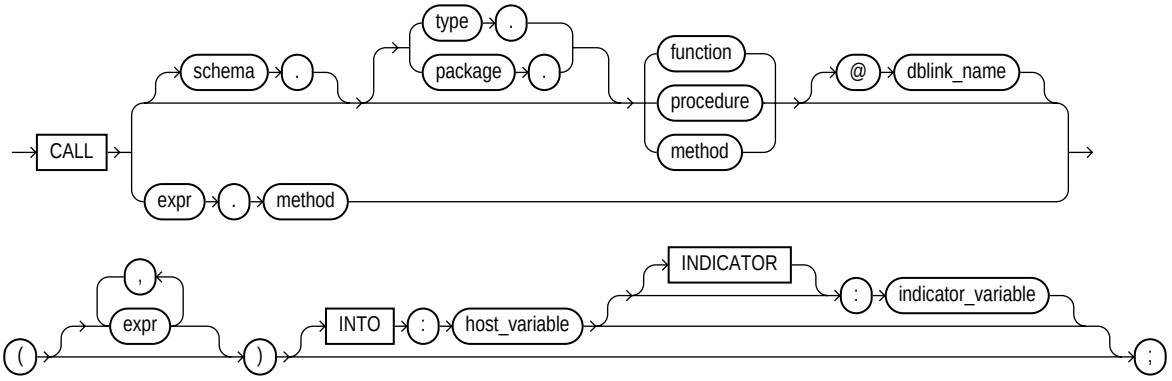
> **See Also:** *PL/SQL User's Guide and Reference* for information on creating such routine

Prerequisites

You must have EXECUTE privilege on the standalone routine or on the type or package in which the routine is defined.

Syntax

call::=



Keywords and Parameters

schema

Specify the schema in which the standalone routine (or the package or type containing the routine) resides. If you do not specify *schema*, then Oracle assumes the routine is in your own schema.

type or *package*

Specify the type or package in which the routine is defined.

function | *procedure* | *method*

Specify the name of the function or procedure being called, or a synonym that translates to a function or procedure.

When you call a type's member function or procedure, if the first argument (SELF) is a null IN OUT argument, then Oracle returns an error. If SELF is a null IN argument, then Oracle returns null. In both cases, the function or procedure is not invoked.

Restriction on functions: If the routine is a function, then the INTO clause is mandatory.

@dblink

In a distributed database system, specify the name of the database containing the standalone routine (or the package or function containing the routine). If you omit *dblink*, then Oracle looks in your local database.

expr

Specify one or more arguments to the routine, if the routine takes arguments.

Restrictions on arguments to the routine:

- An *expr* cannot be a pseudocolumn or either of the object reference functions VALUE or REF.
- Any *expr* that is an IN OUT or OUT argument of the routine must correspond to a host variable expression.

INTO :*host_variable*

The INTO clause applies only to calls to functions. Specify which host variable will store the return value of the function.

:*indicator_variable*

Specify the value or condition of the host variable.

> **See Also:** *Pro*C/C++ Precompiler Programmer's Guide* for more information on host variables and indicator variables

Example

Calling a Procedure: Example The following statement uses the `remove_dept` procedure (created in "Creating a Package Body: Example" on page 14-57) to remove the Entertainment department (created in "Inserting Sequence Values: Example" on page 17-68):

```
CALL remove_dept(280);
```

COMMENT

Purpose

Use the COMMENT statement to add a comment about a table, view, materialized view, or column into the data dictionary.

You can view the comments on a particular table or column by querying the data dictionary views USER_TAB_COMMENTS, DBA_TAB_COMMENTS, or ALL_TAB_COMMENTS or USER_COL_COMMENTS, DBA_COL_COMMENTS, or ALL_COL_COMMENTS.

To drop a comment from the database, set it to the empty string ' '.

See Also:

- "Comments" on page 2-90 for more information on associating comments with SQL statements and schema objects
- *Oracle9i Database Reference* for information on the data dictionary views

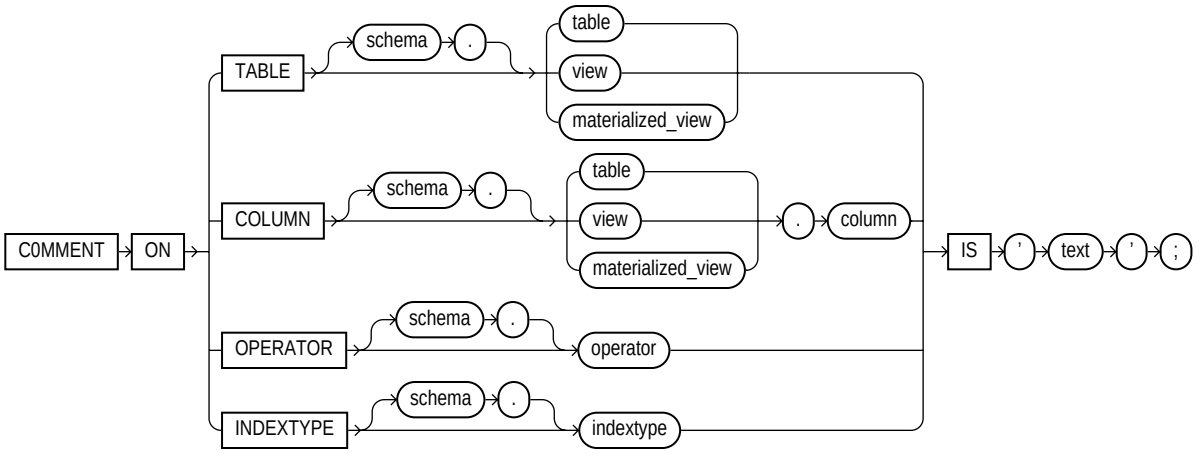
Prerequisites

The object about which you are adding a comment must be in your own schema or:

- You must have COMMENT ANY TABLE system privilege to add a comment to a table, view, or materialized view.
- You must have the COMMENT ANY INDEXTYPE system privilege to add a comment to an indextype.
- You must have the COMMENT ANY OPERATOR system privilege to add a comment to an operator.

Syntax

comment::=



Keywords and Parameters

TABLE Clause

Specify the schema and name of the table, view, or materialized view to be commented. If you omit *schema*, then Oracle assumes the table, view, or materialized view is in your own schema.

COLUMN Clause

Specify the name of the column of a table, view, or materialized view to be commented. If you omit *schema*, then Oracle assumes the table, view, or materialized view is in your own schema.

IS '*text*'

Specify the text of the comment.

> **See Also:** "Text Literals" on page 2-54 for a syntax description of 'text'

OPERATOR Clause

Specify the name of the operator to be commented. If you omit *schema*, then Oracle assumes the operator is in your own schema.

INDEXTYPE Clause

Specify the name of the indextype to be commented. If you omit *schema*, then Oracle assumes the indextype is in your own schema.

Example

Creating Comments: Example To insert an explanatory remark on the `job_id` column of the `employees` table, you might issue the following statement:

```
COMMENT ON COLUMN employees.job_id
   IS 'abbreviated job title';
```

To drop this comment from the database, issue the following statement:

```
COMMENT ON COLUMN employees.job_id IS ' ';
```

COMMIT

Purpose

Use the COMMIT statement to end your current transaction and make permanent all changes performed in the transaction. A **transaction** is a sequence of SQL statements that Oracle treats as a single unit. This statement also erases all savepoints in the transaction and releases the transaction's locks.

Note: Oracle issues an implicit COMMIT before and after any data definition language (DDL) statement.

You can also use this statement to

- Commit an in-doubt distributed transaction manually
- Terminate a read-only transaction begun by a SET TRANSACTION statement.

Oracle Corporation recommends that you explicitly end every transaction in your application programs with a COMMIT or ROLLBACK statement, including the last transaction, before disconnecting from Oracle. If you do not explicitly commit the transaction and the program terminates abnormally, then the last uncommitted transaction is automatically rolled back.

A normal exit from most Oracle utilities and tools causes the current transaction to be committed. A normal exit from an Oracle precompiler program does not commit the transaction and relies on Oracle to roll back the current transaction.

See Also:

- *Oracle9i Database Concepts* for more information on transactions
- SET TRANSACTION on page 18-50 for more information on specifying characteristics of a transaction

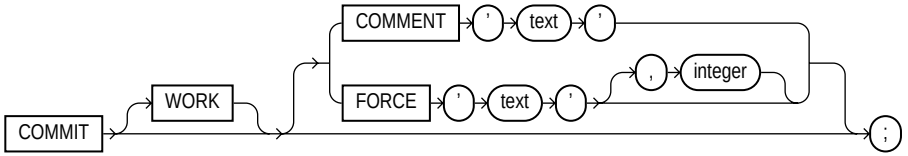
Prerequisites

You need no privileges to commit your current transaction.

To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Syntax

commit::=



Keywords and Parameters

WORK

The WORK keyword is supported for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.

COMMENT Clause

Specify a comment to be associated with the current transaction. The '*text*' is a quoted literal of up to 255 bytes that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in doubt.

> **See Also:** COMMENT on page 12-69 for more information on adding comments to SQL statements

FORCE Clause

In a distributed database system, the FORCE clause lets you manually commit an in-doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can use *integer* to specifically assign the transaction a system change number (SCN). If you omit *integer*, then the transaction is committed using the current SCN.

> **Note:** A COMMIT statement with a FORCE clause commits only the specified transaction. Such a statement does not affect your current transaction.

Restriction on FORCE: COMMIT statements using the FORCE clause are not supported in PL/SQL.

See Also: *Oracle9i Heterogeneous Connectivity Administrator's Guide* for more information on these topics

Examples

Committing an Insert: Example This statement inserts a row into the `hr.regions` table and commits this change:

```
INSERT INTO regions VALUES (5, 'Antarctica');

COMMIT WORK;
```

Commenting on COMMIT: Example The following statement commits the current transaction and associates a comment with it:

```
COMMIT
    COMMENT 'In-doubt transaction Code 36, Call (415) 555-2637';
```

If a network or machine failure prevents this distributed transaction from committing properly, then Oracle stores the comment in the data dictionary along with the transaction ID. The comment indicates the part of the application in which the failure occurred and provides information for contacting the administrator of the database where the transaction was committed.

Forcing an In-Doubt Transaction: Example The following statement manually commits an in-doubt distributed transaction:

```
COMMIT FORCE '22.57.53';
```

13

SQL Statements: CREATE CLUSTER to CREATE JAVA

This chapter contains the following SQL statements:

- CREATE CLUSTER
- CREATE CONTEXT
- CREATE CONTROLFILE
- CREATE DATABASE
- CREATE DATABASE LINK
- CREATE DIMENSION
- CREATE DIRECTORY
- CREATE FUNCTION
- CREATE INDEX
- CREATE INDEXTYPE
- CREATE JAVA

CREATE CLUSTER

Purpose

Use the CREATE CLUSTER statement to create a cluster. A **cluster** is a schema object that contains data from one or more tables, all of which have one or more columns in common. Oracle stores together all the rows (from all the tables) that share the same cluster key.

For information on existing clusters, query the USER_CLUSTERS, ALL_CLUSTERS, and DBA_CLUSTERS data dictionary views.

See Also:

- *Oracle9i Database Concepts* for general information on clusters
- *Oracle9i Application Developer's Guide - Fundamentals* for information on performance considerations of clusters
- *Oracle9i Database Performance Tuning Guide and Reference* for suggestions on when to use clusters
- *Oracle9i Database Reference* for information on the data dictionary views

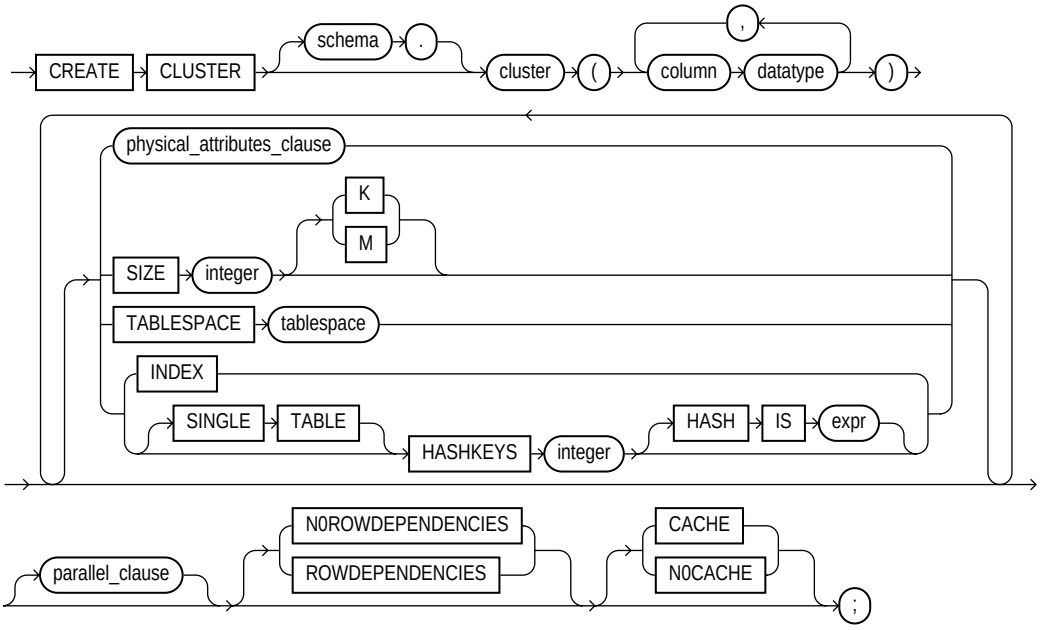
Prerequisites

To create a cluster in your own schema, you must have CREATE CLUSTER system privilege. To create a cluster in another user's schema, you must have CREATE ANY CLUSTER system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or the UNLIMITED TABLESPACE system privilege.

Oracle does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language (DML) statements cannot be issued against clustered tables until you create a cluster index.

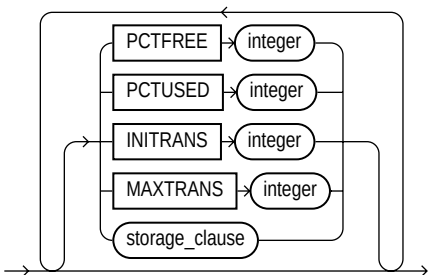
Syntax

create_cluster::=



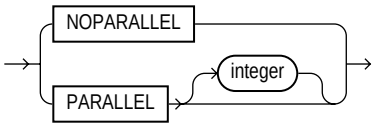
(*physical_attributes_clause::=* on page 13-3)

physical_attributes_clause::=



(*storage_clause* on page 7-56)

parallel_clause::=



Keywords and Parameters

schema

Specify the schema to contain the cluster. If you omit *schema*, Oracle creates the cluster in your current schema.

cluster

Specify is the name of the cluster to be created.

After you create a cluster, you add tables to it. A cluster can contain a maximum of 32 tables. After you create a cluster and add tables to it, the cluster is transparent. You can access clustered tables with SQL statements just as you can access nonclustered tables.

> **See Also:** CREATE TABLE on page 15-7 for information on adding tables to a cluster, Creating a Cluster: Example on page 13-9, and "Adding Tables to a Cluster: Example" on page 13-10

column

Specify one or more names of columns in the cluster key. You can specify up to 16 cluster key columns. These columns must correspond in both datatype and size to columns in each of the clustered tables, although they need not correspond in name.

You cannot specify integrity constraints as part of the definition of a cluster key column. Instead, you can associate integrity constraints with the tables that belong to the cluster.

> **See Also:** "Cluster Keys: Example" on page 13-10

datatype

Specify the datatype of each cluster key column.

Restrictions on datatypes:

- You cannot specify a cluster key column of datatype LONG, LONG RAW, REF, nested table, varray, BLOB, CLOB, BFILE, or user-defined object type.
- You cannot use the HASH IS clause if any column datatype is not INTEGER or NUMBER with scale 0.
- You can specify a column of type ROWID, but Oracle does not guarantee that the values in such columns are valid rowids.

> **See Also:** "Datatypes" on page 2-2 for information on datatypes

physical_attributes_clause

The *physical_attributes_clause* lets you specify the storage characteristics of the cluster. Each table in the cluster uses these storage characteristics as well. If you do not specify values for these parameters, Oracle uses the following defaults:

- PCTFREE: 10
- PCTUSED: 40
- INITRANS: 2 or the default value of the cluster's tablespace, whichever is greater
- MAXTRANS: the default value for the cluster's tablespace

> **See Also:**
>
> - *physical_attributes_clause* on page 7-52 for a complete description of the parameters of the *physical_attributes_clause*
> - *storage_clause* on page 7-56 for a complete description of the *storage_clause*, including default values

SIZE

Specify the amount of space in bytes reserved to store all rows with the same cluster key value or the same hash value. Use K or M to specify this space in kilobytes or megabytes. This space determines the maximum number of cluster or hash values stored in a data block. If SIZE is not a divisor of the data block size, Oracle uses the next largest divisor. If SIZE is larger than the data block size, Oracle uses the operating system block size, reserving at least one data block for each cluster or hash value.

Oracle also considers the length of the cluster key when determining how much space to reserve for the rows having a cluster key value. Larger cluster keys require larger sizes. To see the actual size, query the KEY_SIZE column of the USER_CLUSTERS data dictionary view. (This does not apply to hash clusters, because hash values are not actually stored in the cluster.)

If you omit this parameter, Oracle reserves one data block for each cluster key value or hash value.

TABLESPACE

Specify the tablespace in which the cluster is created.

INDEX Clause

Specify INDEX to create an **indexed cluster**. In an indexed cluster, Oracle stores together rows having the same cluster key value. Each distinct cluster key value is stored only once in each data block, regardless of the number of tables and rows in which it occurs.

After you create an indexed cluster, you must create an index on the cluster key before you can issue any data manipulation language (DML) statements against a table in the cluster. This index is called the **cluster index**.

Note: You cannot create a cluster index for a hash cluster, and you need not create an index on a hash cluster key. If you specify neither INDEX nor HASHKEYS, Oracle creates an indexed cluster by default.

See Also: CREATE INDEX on page 13-62 for information on creating a cluster index and *Oracle9i Database Concepts* for general information in indexed clusters

HASHKEYS Clause

Specify the HASHKEYS clause to create a **hash cluster** and specify the number of hash values for a hash cluster. In a hash cluster, Oracle stores together rows that have the same hash key value. The hash value for a row is the value returned by the cluster's hash function.

Oracle rounds up the HASHKEYS value to the nearest prime number to obtain the actual number of hash values. The minimum value for this parameter is 2. If you

omit both the INDEX clause and the HASHKEYS parameter, Oracle creates an indexed cluster by default.

When you create a hash cluster, Oracle immediately allocates space for the cluster based on the values of the SIZE and HASHKEYS parameters.

> **See Also:** *Oracle9i Database Concepts* for more information on how Oracle allocates space for clusters and "Hash Clusters: Examples" on page 13-10

SINGLE TABLE SINGLE TABLE indicates that the cluster is a type of hash cluster containing only one table. This clause can provide faster access to rows than would result if the table were not part of a cluster.

Restriction on the SINGLE TABLE clause: Only one table can be present in the cluster at a time. However, you can drop the table and create a different table in the same cluster.

> **See Also:** "Single-Table Hash Clusters: Example" on page 13-11

HASH IS *expr* Specify an expression to be used as the hash function for the hash cluster. The expression:

- Must evaluate to a positive value
- Must contain at least one column with referenced columns of any datatype as long as the entire expression evaluates to a number of scale 0. For example: NUM_COLUMN * length(VARCHAR2_COLUMN)
- Cannot reference user-defined PL/SQL functions
- Cannot reference the pseudocolumns LEVEL or ROWNUM
- Cannot reference the user-related functions USERENV, UID, or USER or the datetime functions CURRENT_DATE, CURRENT_TIMESTAMP, DBTIMEZONE, EXTRACT (datetime), FROM_TZ, LOCALTIMESTAMP, NUMTODSINTERVAL, NUMTOYMINTERVAL, SESSIONTIMEZONE, SYSDATE, SYSTIMESTAMP, TO_DSINTERVAL, TO_TIMESTAMP, TO_DATE, TO_TIMESTAMP_TZ, TO_YMINTERVAL, and TZ_OFFSET.
- Cannot evaluate to a constant
- Cannot be a scalar subquery expression
- Cannot contain columns qualified with a schema or object name (other than the cluster name)

If you omit the HASH IS clause, Oracle uses an internal hash function for the hash cluster.

For information on existing hash functions, query the USER_, ALL_, and DBA_CLUSTER_HASH_EXPRESSIONS data dictionary tables.

The cluster key of a hash column can have one or more columns of any datatype. Hash clusters with composite cluster keys or cluster keys made up of noninteger columns must use the internal hash function.

> **See Also:** *Oracle9i Database Reference* for information on the data dictionary views

parallel_clause

The *parallel_clause* lets you parallelize the creation of the cluster.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restriction on the *parallel_clause*: If the tables in *cluster* contain any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *cluster* are executed serially without notification.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

NOROWDEPENDENCIES | ROWDEPENDENCIES

This clause lets you specify whether *cluster* will use **row-level dependency tracking**. With this feature, each row in the tables that make up the cluster has a system change number (SCN) that represents a time greater than or equal to the commit time of the last transaction that modified the row. You cannot change this setting after *cluster* is created.

ROWDEPENDENCIES Specify ROWDEPENDENCIES if you want to enable row-level dependency tracking. This setting is useful primarily to allow for parallel propagation in replication environments. It increases the size of each row by 6 bytes.

NOROWDEPENDENCIES Specify NOROWDEPENDENCIES if you do not want to use the row level dependency tracking feature. This is the default.

> **See Also:** *Oracle9i Replication* for information about the use of row-level dependency tracking in replication environments

CACHE | NOCACHE

CACHE Specify CACHE if you want the blocks retrieved for this cluster to be placed at the *most recently used* end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables.

NOCACHE Specify NOCACHE if you want the blocks retrieved for this cluster to be placed at the *least recently used* end of the LRU list in the buffer cache when a full table scan is performed. This is the default behavior.

Note: NOCACHE has no effect on clusters for which you specify KEEP in the *storage_clause*.

Examples

Creating a Cluster: Example The following statement creates a cluster named personnel with the cluster key column department, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
   (department NUMBER(4))
```

```
SIZE 512
STORAGE (initial 100K next 50K);
```

Cluster Keys: Example The following statement creates the cluster index on the cluster key of `personnel`:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can add tables to the index and perform DML operations on those tables.

Adding Tables to a Cluster: Example The following statements create some departmental tables from the sample `hr.employees` table and add them to the personnel cluster created in the earlier example:

```
CREATE TABLE dept_10
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE department_id = 10;

CREATE TABLE dept_20
   CLUSTER personnel (department_id)
   AS SELECT * FROM employees WHERE department_id = 20;
```

Hash Clusters: Examples The following statement creates a hash cluster named `language` with the cluster key column `cust_language`, a maximum of 10 hash key values, each of which is allocated 512 bytes, and storage parameter values:

```
CREATE CLUSTER language (cust_language VARCHAR2(3))
   SIZE 512 HASHKEYS 10
   STORAGE (INITIAL 100k next 50k);
```

Because the preceding statement omits the `HASH IS` clause, Oracle uses the internal hash function for the cluster.

The following statement creates a hash cluster named `address` with the cluster key made up of the columns `postal_code` and `country_id`, and uses a SQL expression containing these columns for the hash function:

```
CREATE CLUSTER address
   (postal_code NUMBER, country_id CHAR(2))
   HASHKEYS 20
   HASH IS MOD(postal_code + country_id, 101);
```

Single-Table Hash Clusters: Example The following statement creates a single-table hash cluster named `cust_orders` with the cluster key `customer_id` and a maximum of 100 hash key values, each of which is allocated 512 bytes:

```
CREATE CLUSTER cust_orders (customer_id NUMBER(6))
   SIZE 512 SINGLE TABLE HASHKEYS 100;
```

CREATE CONTEXT

Purpose

Use the CREATE CONTEXT statement to:

- Create a namespace for a **context** (a set of application-defined attributes that validates and secures an application) and
- Associate the namespace with the externally created package that sets the context.

You can use the DBMS_SESSION.set_context procedure in your designated package to set or reset the attributes of the context.

See Also:

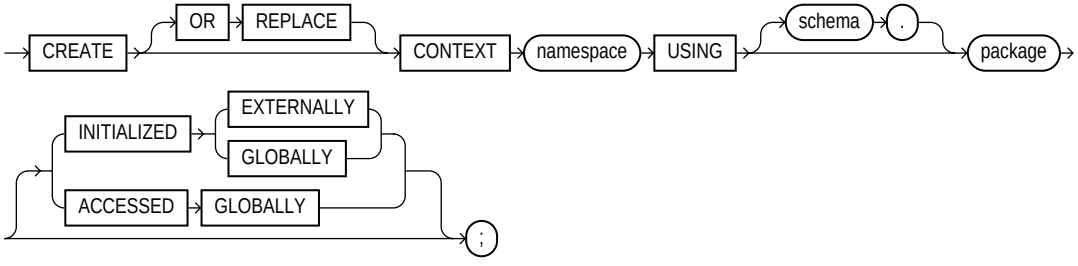
- *Oracle9i Database Concepts* for a definition and discussion of contexts
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the DBMS_SESSION.set_context procedure

Prerequisites

To create a context namespace, you must have CREATE ANY CONTEXT system privilege.

Syntax

create_context::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to redefine an existing context namespace using a different package.

namespace

Specify the name of the context namespace to create or modify. Context namespaces are always stored in the schema SYS.

schema

Specify the schema owning *package*. If you omit *schema*, Oracle uses the current schema.

package

Specify the PL/SQL package that sets or resets the context attributes under the namespace for a user session.

Note: To provide some design flexibility, Oracle does not verify the existence of the schema or the validity of the package at the time you create the context.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information on setting the package

INITIALIZED Clause

The INITIALIZED clause lets you specify an entity other than Oracle that can initialize the context namespace.

EXTERNALLY EXTERNALLY indicates that the namespace can be initialized using an OCI interface when establishing a session.

See Also: *Oracle Call Interface Programmer's Guide* for information on using OCI to establish a session

GLOBALLY GLOBALLY indicates that the namespace can be initialized by the LDAP directory when a global user connects to the database.

After the session is established, only the designated PL/SQL package can issue commands to write to any attributes inside the namespace.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information on establishing globally initialized contexts
- *Oracle Internet Directory Administrator's Guide* for information on the connecting to the database through the LDAP directory

ACCESSED GLOBALLY

This clause indicates that any application context set in *namespace* is accessible throughout the entire instance. This setting lets multiple sessions share application attributes.

Examples

Creating an Application Context: Example This example uses the PL/SQL package `emp_mgmt`, created in "Creating a Package: Example" on page 14-53, which validates and secures the `hr` application. The following statement creates the context namespace `hr_context` and associates it with the package `emp_mgmt`:

```
CREATE CONTEXT hr_context USING emp_mgmt;
```

You can control data access based on this context using the `SYS_CONTEXT` function. For example, suppose your `emp_mgmt` package has defined an attribute `new_empno` as a particular employee identifier. You can secure the base table `employees` by creating a view that restricts access based on the value of `new_empno`, as follows:

```
CREATE VIEW hr_org_secure_view AS
   SELECT * FROM employees
   WHERE employee_id = SYS_CONTEXT('hr_context', 'new_empno');
```

See Also: SYS_CONTEXT on page 6-153

CREATE CONTROLFILE

Caution: Oracle recommends that you perform a full backup of all files in the database before using this statement. For more information, see *Oracle9i User-Managed Backup and Recovery Guide*.

See Also: "BACKUP CONTROLFILE Clause" of ALTER DATABASE on page 9-44 for information creating a script based on an existing database controlfile

Purpose

Use the CREATE CONTROLFILE statement to re-create a control file in one of the following cases:

- All copies of your existing control files have been lost through media failure.
- You want to change the name of the database.
- You want to change the maximum number of redo log file groups, redo log file members, archived redo log files, datafiles, or instances that can concurrently have the database mounted and open.

When you issue a CREATE CONTROLFILE statement, Oracle creates a new control file based on the information you specify in the statement. If you omit any clauses, Oracle uses the default values rather than the values for the previous control file. After successfully creating the control file, Oracle mounts the database in the mode specified by the initialization parameter CLUSTER_DATABASE. You then must perform media recovery before opening the database. It is recommended that you then shut down the instance and take a full backup of all files in the database.

See Also: *Oracle9i User-Managed Backup and Recovery Guide*

Prerequisites

To create a control file, you must have the SYSDBA system privilege.

The database must not be mounted by any instance. Oracle leaves the database mounted in EXCLUSIVE state after successful creation of the control file. If you are using Oracle with Real Application Clusters, the DBA must then shut down and remount the database in SHARED mode (which is the default if the value of the

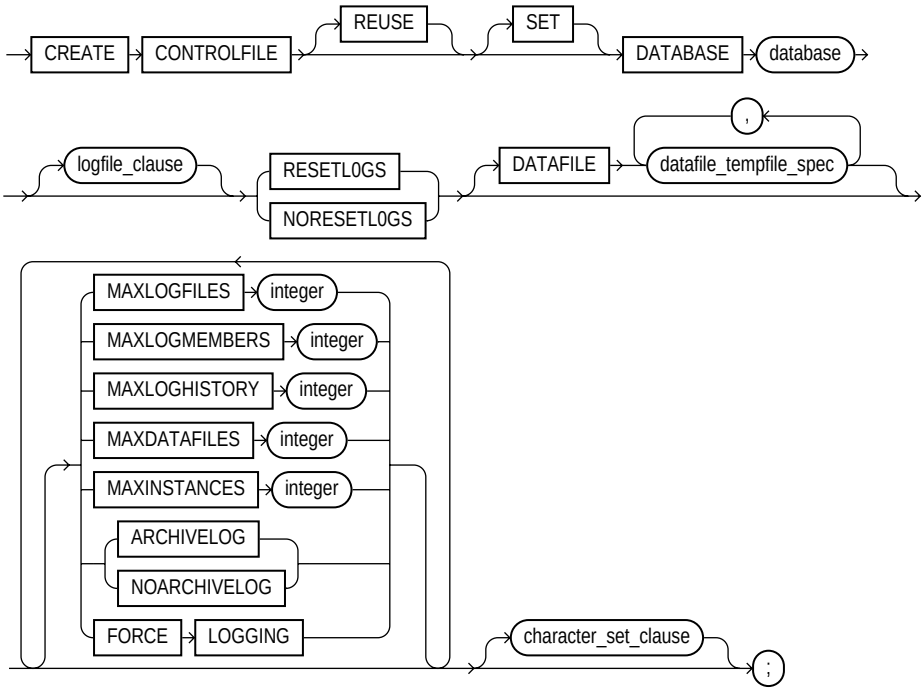
CLUSTER_DATABASE initialization parameter is TRUE) before other instances can start up.

If the REMOTE_LOGIN_PASSWORDFILE initialization parameter is set to EXCLUSIVE, Oracle returns an error when you attempt to re-create the control file. To avoid this message, either set the parameter to SHARED, or re-create your password file before re-creating the control file.

> **See Also:** *Oracle9i Database Reference* for more information about the REMOTE_LOGIN_PASSWORDFILE parameter

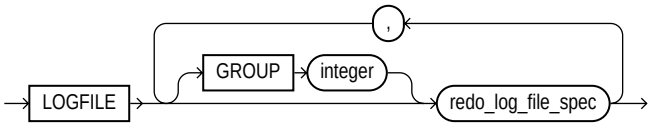
Syntax

create_controlfile::=



(*datafile_tempfile_spec::=* on page 7-39—part of *file_specification* syntax)

logfile_clause::=



(*redo_log_file_spec::=* on page 7-40—part of *file_specification* syntax)

character_set_clause::=

Keywords and Parameters

REUSE

Specify REUSE to indicate that existing control files identified by the initialization parameter CONTROL_FILES can be reused, thus ignoring and overwriting any information they may currently contain. If you omit this clause and any of these control files already exists, Oracle returns an error.

DATABASE Clause

Specify the name of the database. The value of this parameter must be the existing database name established by the previous CREATE DATABASE statement or CREATE CONTROLFILE statement.

SET DATABASE Clause

Use SET DATABASE to change the name of the database. The name of a database can be as long as eight bytes.

logfile_clause

Use the *logfile_clause* to specify the redo log files for your database. You must list all members of all redo log file groups.

GROUP *integer* Specify the logfile group number. If you specify GROUP values, Oracle verifies these values with the GROUP values when the database was last open.

If you omit this clause, Oracle creates logfiles using system default values. In addition, if either the DB_CREATE_ONLINE_LOG_DEST_n or DB_CREATE_FILE_DEST initialization parameter (or both) has been set, and if you have specified RESETLOGS, then Oracle creates two logs in the default logfile destination specified in the DB_CREATE_ONLINE_LOG_DEST_*n* parameter, and if it is not set, then in the DB_CREATE_FILE_DEST parameter.

> **See Also:** *file_specification* on page 7-39 for a full description of this clause

RESETLOGS Specify RESETLOGS if you want Oracle to ignore the contents of the files listed in the LOGFILE clause. These files do not have to exist. Each *redo_log_file_spec* in the LOGFILE clause must specify the SIZE parameter. Oracle assigns all online redo log file groups to thread 1 and enables this thread for public use by any instance. After using this clause, you must open the database using the RESETLOGS clause of the ALTER DATABASE statement.

NORESETLOGS Specify NORESETLOGS if you want Oracle to use all files in the LOGFILE clause as they were when the database was last open. These files must exist and must be the current online redo log files rather than restored backups. Oracle reassigns the redo log file groups to the threads to which they were previously assigned and reenables the threads as they were previously enabled.

DATAFILE Clause

Specify the datafiles of the database. You must list all datafiles. These files must all exist, although they may be restored backups that require media recovery. See the syntax description in *file_specification* on page 7-39.

> **Note:** You should list only datafiles in this clause, not temporary datafiles (tempfiles). Please refer to *Oracle9i User-Managed Backup and Recovery Guide* for more information on handling tempfiles.

Restriction on DATAFILE: You cannot specify the *autoextend_clause* of *data_file_spec* in this DATAFILE clause.

MAXLOGFILES Clause

Specify the maximum number of online redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default and maximum values

depend on your operating system. The value that you specify should not be less than the greatest GROUP value for any redo log file group.

MAXLOGMEMBERS Clause

Specify the maximum number of members, or identical copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY Clause

This parameter is useful only if you are using Oracle in ARCHIVELOG mode with Real Application Clusters. Specify the maximum number of archived redo log file groups for automatic media recovery of Real Application Clusters. Oracle uses this value to determine how much space in the control file to allocate for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file.

MAXDATAFILES Clause

Specify the initial sizing of the datafiles section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the control file to expand automatically so that the datafiles section can accommodate more files.

The number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.

MAXINSTANCES Clause

Specify the maximum number of instances that can simultaneously have the database mounted and open. This value takes precedence over the value of the initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.

ARCHIVELOG | NOARCHIVELOG

Specify ARCHIVELOG to archive the contents of redo log files before reusing them. This clause prepares for the possibility of media recovery as well as instance or system failure recovery.

If you omit both the ARCHIVELOG clause and NOARCHIVELOG clause, Oracle chooses NOARCHIVELOG mode by default. After creating the control file, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.

FORCE LOGGING

Use this clause to put the database into FORCE LOGGING mode after control file creation. When the database is in this mode, Oracle logs all changes in the database except changes to temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any NOLOGGING or FORCE LOGGING settings you specify for individual tablespaces and any NOLOGGING settings you specify for individual database objects. If you omit this clause, the database will not be in FORCE LOGGING mode after the controlfile is created.

Note: FORCE LOGGING mode can have performance effects. Please refer to *Oracle9i Database Administrator's Guide* for information on when to use this setting.

---

character_set_clause

If you specify a character set, Oracle reconstructs character set information in the control file. In case media recovery of the database is required, this information will be available before the database is open, so that tablespace names can be correctly interpreted during recovery. This clause is required only if you are using a character set other than the default US7ASCII. Oracle prints the current database character set to the "alert" log in $ORACLE_HOME/log during startup.

If you are re-creating your control file and you are using Recovery Manager for tablespace recovery, and if you specify a different character set from the one stored in the data dictionary, then tablespace recovery will not succeed. (However, at database open, the control file character set will be updated with the correct character set from the data dictionary.)

Note: You cannot modify the character set of the database with this clause.

See Also: *Oracle9i Recovery Manager User's Guide* for more information on tablespace recovery

Example

Creating a Controlfile: Example This statement re-creates a control file. In this statement, database demo was created with the WE8DEC character set. The example uses the word *path* where you would normally insert the path on your system to the appropriate Oracle directories.

```
STARTUP NOMOUNT

CREATE CONTROLFILE REUSE DATABASE "demo" NORESETLOGS NOARCHIVELOG
    MAXLOGFILES 32
    MAXLOGMEMBERS 2
    MAXDATAFILES 32
    MAXINSTANCES 1
    MAXLOGHISTORY 449
LOGFILE
  GROUP 1 '/path/oracle/dbs/t_log1.f'  SIZE 500K,
  GROUP 2 '/path/oracle/dbs/t_log2.f'  SIZE 500K
# STANDBY LOGFILE
DATAFILE
  '/path/oracle/dbs/t_db1.f',
  '/path/oracle/dbs/dbu19i.dbf',
  '/path/oracle/dbs/tbs_11.f',
  '/path/oracle/dbs/smundo.dbf',
  '/path/oracle/dbs/demo.dbf'
CHARACTER SET WE8DEC
;
```

CREATE DATABASE

Caution: This statement prepares a database for initial use and erases any data currently in the specified files. Use this statement only when you understand its ramifications.

Note Regarding Security Enhancements: In this release of Oracle and in subsequent releases, several enhancements are being made to ensure the security of default database user accounts.

- During initial installation with the Database Configuration Assistant (DCBA), all default database user accounts except SYS, SYSTEM, SCOTT, DBSNMP, OUTLN, AURORAJISUTILITY$, AURORA$ORB$UNAUTHENTICATED and OSE$HTTP$ADMIN are locked and expired. To activate a locked account, the DBA must manually unlock it and reassign it a new password.
- In addition, the DBCA prompts for passwords for users SYS and SYSTEM during initial installation of the database rather than assigning default passwords to them. A CREATE DATABASE statement issued manually also lets you supply passwords for these two users.

Purpose

Use the CREATE DATABASE statement to create a database, making it available for general use.

This statement erases all data in any specified datafiles that already exist in order to prepare them for initial database use. If you use the statement on an existing database, all data in the datafiles is lost.

After creating the database, this statement mounts it in either exclusive or parallel mode (depending on the value of the CLUSTER_DATABASE initialization parameter) and opens it, making it available for normal use. You can then create tablespaces and rollback segments for the database.

See Also:

- ALTER DATABASE on page 9-13 for information on modifying a database
- *Oracle9i Java Developer's Guide* for information on creating an Oracle9*i* Java virtual machine
- CREATE ROLLBACK SEGMENT on page 14-80 and CREATE TABLESPACE on page 15-80 for information on creating rollback segments and tablespaces

Prerequisites

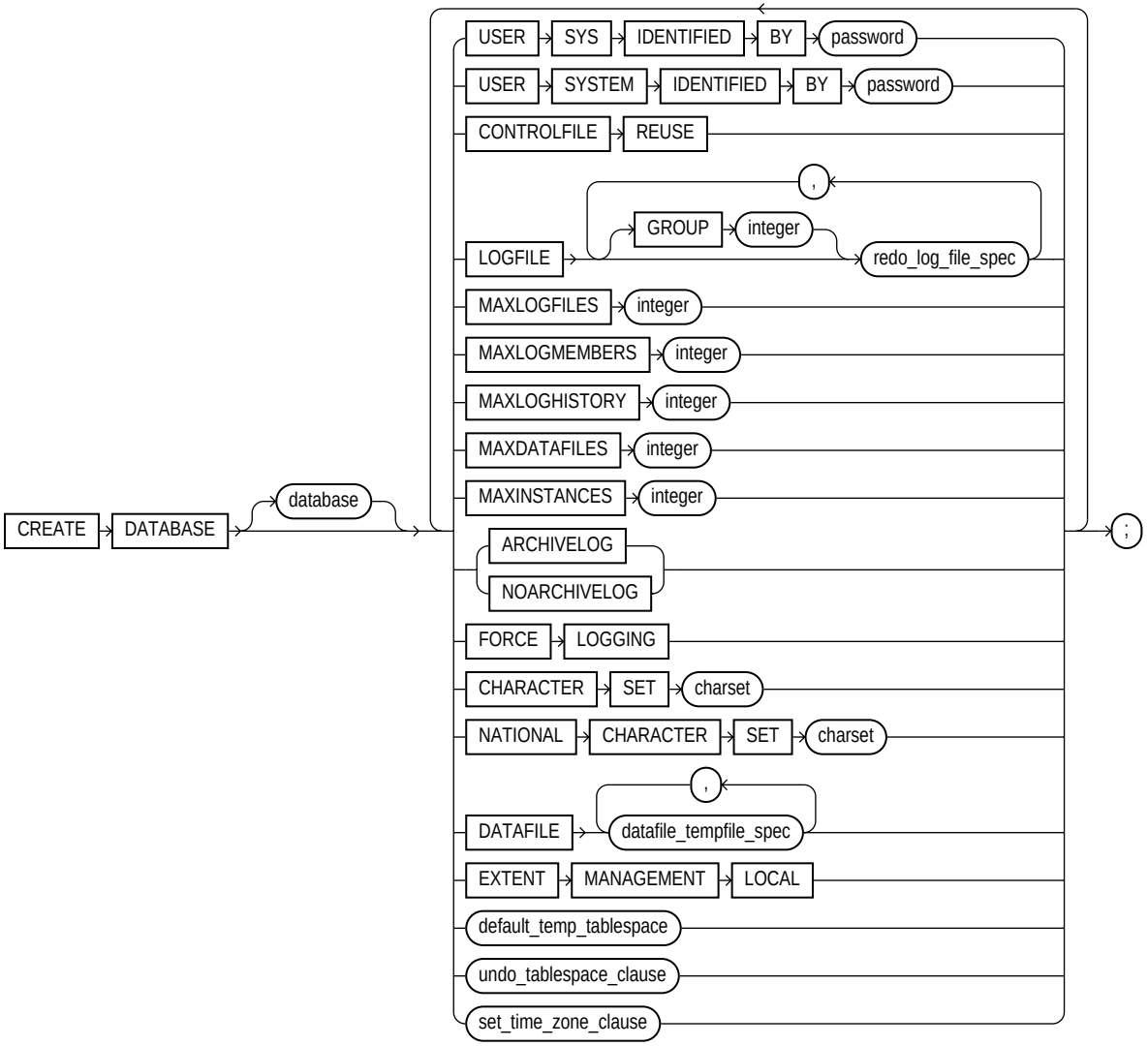
To create a database, you must have the SYSDBA system privilege.

If the REMOTE_LOGIN_PASSWORDFILE initialization parameter is set to EXCLUSIVE, Oracle returns an error when you attempt to re-create the database. To avoid this message, either set the parameter to SHARED, or re-create your password file before re-creating the database.

See Also: *Oracle9i Database Reference* for more information about the REMOTE_LOGIN_PASSWORDFILE parameter

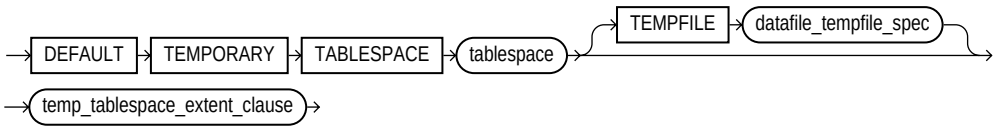
Syntax

create_database::=



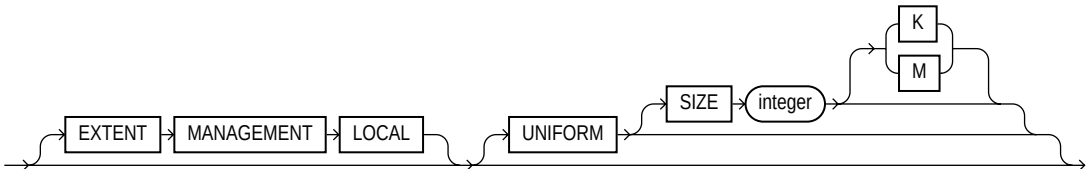
(*redo_log_file_spec::=* on page 7-40, *datafile_tempfile_spec::=* on page 7-39, *default_temp_tablespace::=* on page 13-25, *undo_tablespace_clause::=* on page 13-25, *set_time_zone_clause::=* on page 13-25)

default_temp_tablespace::=

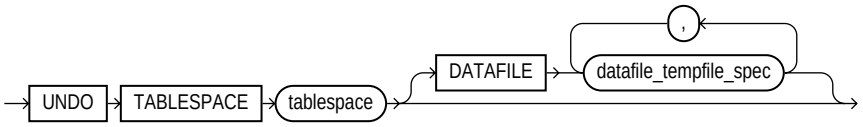


(*datafile_tempfile_spec::=* on page 7-39—part of *file_specification*)

temp_tablespace_extent::=

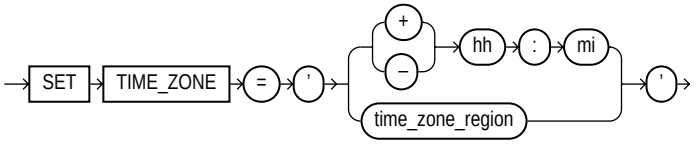


undo_tablespace_clause::=



(*datafile_tempfile_spec::=* on page 7-39—part of *file_specification*)

set_time_zone_clause::=



Keyword and Parameters

database

Specify the name of the database to be created. The name can be up to 8 bytes long. The database name can contain only ASCII characters. Oracle writes this name into the control file. If you subsequently issue an ALTER DATABASE statement that explicitly specifies a database name, Oracle verifies that name with the name in the control file.

Note: You cannot use special characters from European or Asian character sets in a database name. For example, characters with umlauts are not allowed.

If you omit the database name from a CREATE DATABASE statement, Oracle uses the name specified by the initialization parameter DB_NAME. If the DB_NAME initialization parameter has been set, and you specify a different name from the value of that parameter, Oracle returns an error.

See Also: "Schema Object Naming Guidelines" on page 2-115 for additional rules to which database names should adhere

USER SYS ..., USER SYSTEM ...

Use these clauses to establish passwords for the SYS and SYSTEM users. These clauses are not mandatory in this release of Oracle9i. However, if you specify either clause, you must specify both clauses.

If you do not specify these clauses, Oracle creates default passwords "change_on_install" for user SYS and "manager" for user SYSTEM. You can subsequently change these passwords using the ALTER USER statement. You can also use ALTER USER to add password management attributes after database creation.

See Also: ALTER USER on page 12-21

CONTROLFILE REUSE Clause

Specify CONTROLFILE REUSE to reuse existing control files identified by the initialization parameter CONTROL_FILES, thus ignoring and overwriting any information they currently contain. Normally you use this clause only when you are re-creating a database, rather than creating one for the first time. You cannot use this clause if you also specify a parameter value that requires that the control file be larger than the existing files. These parameters are MAXLOGFILES, MAXLOGMEMBERS, MAXLOGHISTORY, MAXDATAFILES, and MAXINSTANCES.

If you omit this clause and any of the files specified by CONTROL_FILES already exist, Oracle returns an error.

LOGFILE Clause

Specify one or more files to be used as redo log files. Each *redo_log_file_spec* specifies a redo log file group containing one or more redo log file members

(copies). All redo log files specified in a CREATE DATABASE statement are added to redo log thread number 1.

> **See Also:** *file_specification* on page 7-39 for a full description

GROUP *integer* Specify the number that identifies the redo log file group. The value of *integer* can range from 1 to the value of the MAXLOGFILES parameter. A database must have at least two redo log file groups. You cannot specify multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.

If you omit the LOGFILE clause:

- If either the DB_CREATE_ONLINE_LOG_DEST_*n* or DB_CREATE_FILE_DEST initialization parameter (or both) is set, then Oracle creates two Oracle-managed logfiles with system-generated names, 100 MB in size, in the default logfile directory specified in the DB_CREATE_ONLINE_LOG_DEST_*n* parameter, and if it is not set, then in the DB_CREATE_FILE_DEST parameter.
- If neither of these parameters is set, Oracle creates two redo log file groups. The names and sizes of the default files depend on your operating system.

MAXLOGFILES Clause

Specify the maximum number of redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default, minimum, and maximum values depend on your operating system.

MAXLOGMEMBERS Clause

Specify the maximum number of members, or copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.

MAXLOGHISTORY Clause

This parameter is useful only if you are using Oracle in ARCHIVELOG mode with Real Application Clusters. Specify the maximum number of archived redo log files for automatic media recovery Real Application Clusters. Oracle uses this value to determine how much space in the control file to allocate for the names of archived

redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file.

MAXDATAFILES Clause

Specify the initial sizing of the datafiles section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the Oracle control file to expand automatically so that the datafiles section can accommodate more files.

The number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.

MAXINSTANCES Clause

Specify the maximum number of instances that can simultaneously have this database mounted and open. This value takes precedence over the value of initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.

ARCHIVELOG | NOARCHIVELOG

ARCHIVELOG Specify ARCHIVELOG if you want the contents of a redo log file group to be archived before the group can be reused. This clause prepares for the possibility of media recovery.

NOARCHIVELOG Specify NOARCHIVELOG if the contents of a redo log file group need not be archived before the group can be reused. This clause does not allow for the possibility of media recovery.

The default is NOARCHIVELOG mode. After creating the database, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.

FORCE LOGGING

Use this clause to put the database into FORCE LOGGING mode. Oracle will log all changes in the database except for changes in temporary tablespaces and temporary segments. This setting takes precedence over and is independent of any NOLOGGING or FORCE LOGGING settings you specify for individual tablespaces and any NOLOGGING settings you specify for individual database objects.

FORCE LOGGING mode is persistent across instances of the database. That is, if you shut down and restart the database, the database is still in FORCE LOGGING mode. However, if you re-create the control file, Oracle will take the database out of FORCE LOGGING mode unless you specify FORCE LOGGING in the CREATE CONTROLFILE statement.

Note: FORCE LOGGING mode can have performance effects. Please refer to *Oracle9i Database Administrator's Guide* for information on when to use this setting.

See Also: CREATE CONTROLFILE on page 13-15

CHARACTER SET Clause

Specify the character set the database uses to store data. The supported character sets and default value of this parameter depend on your operating system.

Restriction on CHARACTER SET: You cannot specify the AL16UTF16 character set as the database character set.

See Also: *Oracle9i Database Globalization Support Guide* for more information about choosing a character set

NATIONAL CHARACTER SET Clause

Specify the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2 (either AF16UTF16 or UTF8). The default is 'AL16UTF16'.

See Also: *Oracle9i Database Globalization Support Guide* for information on Unicode datatype support.

DATAFILE Clause

Specify one or more files to be used as datafiles. All these files become part of the SYSTEM tablespace.

If you are running the database in automatic undo mode and you specify a datafile name for the SYSTEM tablespace, then Oracle expects to generate datafiles for all tablespaces. Oracle does this automatically if you are using Oracle-managed files (that is, you have set values for the DB_CREATE_FILE_DEST or DB_CREATE_ONLINE_LOG_DEST_*n* initialization parameter). However, if you are not using

Oracle-managed files and you specify this clause, then you must also specify the *undo_tablespace_clause* and the *default_temp_tablespace* clause.

If you omit this clause:

- If the DB_CREATE_FILE_DEST initialization parameter is set, Oracle creates a 100 MB Oracle-managed datafile with a system-generated name in the default file destination specified in the parameter.
- If the DB_CREATE_FILE_DEST initialization parameter is not set, Oracle creates one datafile whose name and size depend on your operating system.

Note: Oracle recommends that the total initial space allocated for the SYSTEM tablespace be a minimum of 5 megabytes.

See Also: *file_specification* on page 7-39 for syntax

EXTENT MANAGEMENT LOCAL

Use this clause to create a locally managed SYSTEM tablespace. If you omit this clause, the SYSTEM tablespace will be dictionary managed.

Caution: Once you create a locally managed SYSTEM tablespace, you cannot change it to be dictionary managed, nor can you create any other dictionary-managed tablespaces in this database.

If you specify this clause, the database must have a default temporary tablespace, because a locally managed SYSTEM tablespace cannot store temporary segments.

- If you specify EXTENT MANAGEMENT LOCAL but you do not specify the DATAFILE clause, you can omit the *default_temp_tablespace* clause. Oracle will create a default temporary tablespace called TEMP with one datafile of size 10M with autoextend disabled.
- If you specify both EXTENT MANAGEMENT LOCAL and the DATAFILE clause, then you must also specify the *default_temp_tablespace* clause and explicitly specify a datafile for that tablespace.

If you have opened the instance in Automatic Undo Management mode, similar requirements exist for the database undo tablespace:

- If you specify EXTENT MANAGEMENT LOCAL but you do not specify the DATAFILE clause, you can omit the *undo_tablespace_clause*. Oracle will create an undo tablespace named SYS_UNDOTBS.
- If you specify both EXTENT MANAGEMENT LOCAL and the DATAFILE clause, then you must also specify the *undo_tablespace_clause* and explicitly specify a datafile for that tablespace.

> **See Also:** *Oracle9i Database Administrator's Guide* for more information on locally managed and dictionary-managed tablespaces

default_temp_tablespace

Specify this clause to create a default temporary tablespace for the database. Oracle will assign to this temporary tablespace any users for whom you do not specify a different temporary tablespace. If you do not specify this clause, the SYSTEM tablespace is the default temporary tablespace.

The TEMPFILE clause part of this clause is optional if you have enabled Oracle-managed files by setting the DB_CREATE_FILE_DEST initialization parameter. If you have not specified a value for this parameter, the TEMPFILE clause is required.

> **Note:** On some operating systems, Oracle does not allocate space for the tempfile until the tempfile blocks are actually accessed. This delay in space allocation results in faster creation and resizing of tempfiles, but it requires that sufficient disk space is available when the tempfiles are later used. Please refer to the Oracle documentation for your operating system to determine whether Oracle allocates tempfile space in this way on your system.

Restrictions on default temporary tablespaces:

- You cannot specify the SYSTEM tablespace in this clause.
- The default temporary tablespace must have a standard block size.

The *temp_tablespace_extent* clause lets you specify how the tablespace is managed.

EXTENT MANAGEMENT LOCAL This clause indicates that some part of the tablespace is set aside for a bitmap. All temporary tablespaces have locally managed extents, so this clause is optional.

UNIFORM *integer* Specify the size of the extents of the temporary tablespace in bytes. All extents of temporary tablespaces are the same size (uniform). If you do not specify this clause, Oracle uses uniform extents of 1M.

SIZE *integer* Specify in bytes the size of the tablespace extents. Use K or M to specify the size in kilobytes or megabytes.

If you do not specify SIZE, Oracle uses the default extent size of 1M.

> **See Also:** *Oracle9i Database Concepts* for a discussion of locally managed tablespaces

undo_tablespace_clause

If you have opened the instance in automatic undo mode (that is, the UNDO_MANAGEMENT initialization parameter is set to AUTO), you can specify the *undo_tablespace_clause* to create a tablespace to be used for undo data. If you want undo space management to be handled by way of rollback segments, omit this clause. You can also omit this clause if you have set a value for the UNDO_TABLESPACE initialization parameter. If that parameter has been set, and if you specify this clause, then *tablespace* must be the same as that parameter value.

The DATAFILE clause part of this clause is optional if you have enabled Oracle Managed Files by setting the DB_CREATE_FILE_DEST initialization parameter. If you have not specified a value for this parameter, the DATAFILE clause is required.

- If you specify this clause, Oracle creates an undo tablespace named *tablespace*, creates the specified datafiles as part of the undo tablespace, and assigns this tablespace as the undo tablespace of the instance. Oracle will handle management of undo data using this undo tablespace. The DATAFILE clause of this clause has the same behavior as described in "DATAFILE Clause" on page 13-29.

> **Note:** If you have specified a value for the UNDO_TABLESPACE initialization parameter in your initialization parameter file before mounting the database, be sure you specify the same name in this clause. If these names differ, Oracle will return an error when you open the database.

- If you omit this clause, Oracle creates a default database with a default undo tablespace named SYS_UNDOTBS and assigns this default tablespace as the undo tablespace of the instance. This undo tablespace allocates disk space from the default files used by the CREATE DATABASE statement, and has an initial extent of 10M. Oracle handles the system-generated datafile as described in "DATAFILE Clause" on page 13-29. If Oracle is unable to create the undo tablespace, the entire CREATE DATABASE operation fails.

  See Also:

  - *Oracle9i Database Reference* for information on opening a database instance in Automatic Undo Management mode using the UNDO_MANAGEMENT parameter
  - *Oracle9i Database Administrator's Guide* for information on Automatic Undo Management and undo tablespaces
  - CREATE TABLESPACE on page 15-80 for information on creating an undo tablespace after database creation

set_time_zone_clause

Use the SET TIME_ZONE clause to set the time zone of the database. You can specify the time zone in two ways:

- By specifying a displacement from UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The valid range of *hh:mm* is -12:00 to +14:00.
- By specifying a time zone region. To see a listing of valid region names, query the TZNAME column of the V$TIMEZONE_NAMES dynamic performance view.

  See Also: *Oracle9i Database Reference* for information on the dynamic performance views

Oracle normalizes all TIMESTAMP WITH LOCAL TIME ZONE data to the time zone of the database when the data is stored on disk. If you do not specify the SET TIME_ZONE clause, Oracle uses the operating system's time zone of the server. If the operating system time zone is not a valid Oracle time zone, the database time zone defaults to UTC.

Examples

Creating a Database: Example The following statement creates a database and fully specifies each argument:

```
CREATE DATABASE sample
   CONTROLFILE REUSE
   LOGFILE
      GROUP 1 ('diskx:log1.log', 'disky:log1.log') SIZE 50K,
      GROUP 2 ('diskx:log2.log', 'disky:log2.log') SIZE 50K
   MAXLOGFILES 5
   MAXLOGHISTORY 100
   MAXDATAFILES 10
   MAXINSTANCES 2
   ARCHIVELOG
   CHARACTER SET AL32UTF8
   NATIONAL CHARACTER SET AL16UTF16
   DATAFILE
      'disk1:df1.dbf' AUTOEXTEND ON,
      'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED
   DEFAULT TEMPORARY TABLESPACE temp_ts
   UNDO TABLESPACE undo_ts
   SET TIME_ZONE = '+02:00';
```

This example assumes that you have enabled Oracle Managed Files by specifying a value for the DB_CREATE_FILE_DEST parameter in your initialization parameter file. Therefore no file specification is needed for the DEFAULT TEMPORARY TABLESPACE and UNDO TABLESPACE clauses.

CREATE DATABASE LINK

Purpose

Use the CREATE DATABASE LINK statement to create a database link. A **database link** is a schema object in the local database that enables you to access objects on a remote database. The remote database need not be an Oracle system.

Once you have created a database link, you can use it to refer to tables and views on the remote database. You can refer to a remote table or view in a SQL statement by appending *@dblink* to the table or view name. You can query a remote table or view with the SELECT statement. If you are using Oracle with the distributed option, you can also access remote tables and views using any INSERT, UPDATE, DELETE, or LOCK TABLE statement.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for information about accessing remote tables or views with PL/SQL functions, procedures, packages, and datatypes
- *Oracle9i Database Administrator's Guide* for information on distributed database systems
- *Oracle9i Database Reference* for descriptions of existing database links in the ALL_DB_LINKS, DBA_DB_LINKS, and USER_DB_LINKS data dictionary views and to monitor the performance of existing links through the V$DBLINK dynamic performance view
- DROP DATABASE LINK on page 16-70 for information on dropping existing database links
- INSERT on page 17-54, UPDATE on page 18-59, DELETE on page 16-55, and LOCK TABLE on page 17-74 for using links in DML operations

Prerequisites

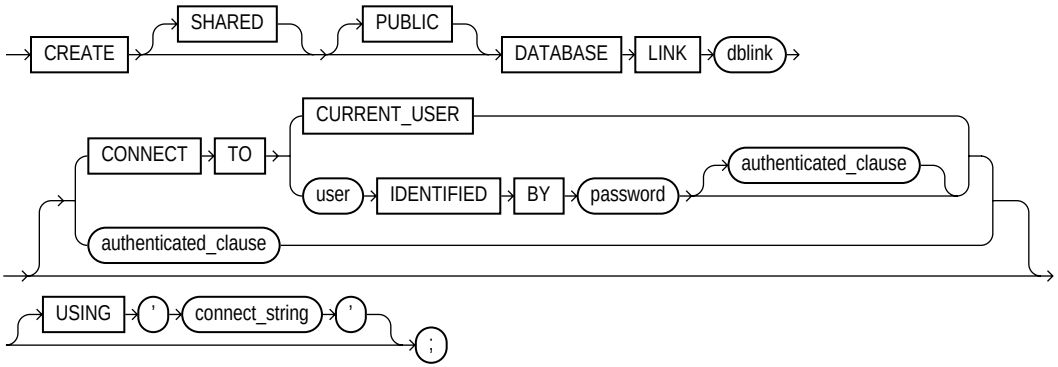
To create a private database link, you must have CREATE DATABASE LINK system privilege. To create a public database link, you must have CREATE PUBLIC DATABASE LINK system privilege. Also, you must have CREATE SESSION privilege on the remote Oracle database.

Oracle Net must be installed on both the local and remote Oracle databases.

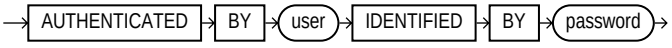
To access non-Oracle systems you must use Oracle Heterogeneous Services.

Syntax

create_database_link::=



authenticated_clause::=



Keyword and Parameters

SHARED

Specify SHARED to use a single network connection to create a public database link that can be shared between multiple users.

> **See Also:** *Oracle9i Heterogeneous Connectivity Administrator's Guide* for more information about shared database links

PUBLIC

Specify PUBLIC to create a public database link available to all users. If you omit this clause, the database link is private and is available only to you.

> **See Also:** "Defining a Public Database Link: Example" on page 13-39

dblink

Specify the complete or partial name of the database link. The value of the GLOBAL_NAMES initialization parameter determines whether the database link must have the same name as the database to which it connects.

The maximum number of database links that can be open in one session or one instance of a Real Application Clusters configuration depends on the value of the OPEN_LINKS and OPEN_LINKS_PER_INSTANCE initialization parameters.

Restriction on database links: You cannot create a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema. (Periods are permitted in names of database links, so Oracle interprets the entire name, such as ralph.linktosales, as the name of a database link in your schema rather than as a database link named linktosales in the schema ralph.)

See Also:

- "Referring to Objects in Remote Databases" on page 2-118 for guidelines for naming database links
- *Oracle9i Database Reference* for information on the GLOBAL_NAMES, OPEN_LINKS, and OPEN_LINKS_PER_INSTANCE initialization parameters

CONNECT TO Clause

The CONNECT TO clause lets you enable a connection to the remote database.

CURRENT_USER Clause

Specify CURRENT_USER to create a **current user database link**. The current user must be a global user with a valid account on the remote database for the link to succeed.

If the database link is used directly, that is, not from within a stored object, then the current user is the same as the connected user.

When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, CURRENT_USER is the username that owns the stored object, and not the username that called the object. For example, if the database link appears inside procedure scott.p (created by scott), and user jane calls procedure scott.p, the current user is scott.

However, if the stored object is an invoker-rights function, procedure, or package, the invoker's authorization ID is used to connect as a remote user. For example, if the privileged database link appears inside procedure scott.p (an invoker-rights

procedure created by `scott`), and user Jane calls procedure `scott.p`, then `CURRENT_USER` is `jane` and the procedure executes with Jane's privileges.

See Also:

- CREATE FUNCTION on page 13-49 for more information on invoker-rights functions
- "Defining a CURRENT_USER Database Link: Example" on page 13-38

user IDENTIFIED BY *password*

Specify the username and password used to connect to the remote database using a **fixed user database link**. If you omit this clause, the database link uses the username and password of each user who is connected to the database. This is called a **connected user database link**.

See Also: "Defining a Fixed-User Database Link: Example" on page 13-39

authenticated_clause

Specify the username and password on the target instance. This clause authenticates the user to the remote server and is required for security. The specified username and password must be a valid username and password on the remote instance. The username and password are used only for authentication. No other operations are performed on behalf of this user.

You must specify this clause when using the SHARED clause.

USING '*connect string*'

Specify the service name of a remote database.

See Also: *Oracle9i Net Services Administrator's Guide* for information on specifying remote databases

Examples

Defining a CURRENT_USER Database Link: Example The following statement defines a current-user database link using the sample database:

```
CREATE DATABASE LINK sales.hq.acme.com
   CONNECT TO CURRENT_USER
   USING 'sales';
```

Defining a Fixed-User Database Link: Example The following statement defines a fixed-user database link named `sales.hq.acme.com`:

```
CREATE DATABASE LINK sales.hq.acme.com
   CONNECT TO hr IDENTIFIED BY hr
   USING 'sales';
```

Once this database link is created, you can query tables in the schema `scott` on the remote database in this manner:

```
SELECT * FROM employees@sales.hq.acme.com;
```

You can also use DML statements to modify data on the remote database:

```
INSERT INTO orders@sales.hq.acme.com
   (customer_id, order_id, order_total)
   VALUES (5001, 1235, 2000);

UPDATE orders@sales.hq.acme.com
   SET order_total = order_total + 500;

DELETE FROM order_id@sales.hq.acme.com
   WHERE order_id = 2443;
```

You can also access tables owned by other users on the same database. This statement assumes that the current user has `SELECT` privileges on the `hr.departments` table:

```
SELECT *
   FROM hr.departments@sales.hq.acme.com;
```

The previous statement connects to the user `scott` on the remote database and then queries Adam's `dept` table.

You can create a synonym to hide the fact that the `departments` table is on a remote database. The following statement causes all future references to `hr_depts` to access a remote `departments` table owned by `hr`:

```
CREATE SYNONYM hr_depts
   FOR hr.departments@sales.hq.acme.com;
```

Defining a Public Database Link: Example The following statement defines a shared public fixed user database link named `sales.hq.acme.com` that refers to user `hr` on the database specified by the string service name '`sales`':

```
CREATE SHARED PUBLIC DATABASE LINK sales.hq.acme.com
   CONNECT TO hr IDENTIFIED BY hr
```

```
AUTHENTICATED BY anupam IDENTIFIED BY bhide
USING 'sales';
```

CREATE DIMENSION

Purpose

Use the CREATE DIMENSION statement to create a **dimension**. A dimension defines a parent-child relationship between pairs of column sets, where all the columns of a column set must come from the same table. However, columns in one column set (or "level") can come from a different table than columns in another set. The optimizer uses these relationships with materialized views to perform **query rewrite**. The Summary Advisor uses these relationships to recommend creation of specific materialized views.

Note: Oracle does not automatically validate the relationships you declare when creating a dimension. To validate the relationships specified in the *hierarchy_clause* and the *join_clause* of CREATE DIMENSION, you must run the DBMS_OLAP.validate_dimension procedure.

See Also:

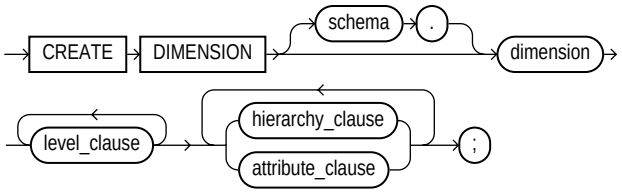
- CREATE MATERIALIZED VIEW on page 14-5 for more information on materialized views
- *Oracle9i Data Warehousing Guide* for more information on query rewrite, the optimizer and the Summary Advisor
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the DBMS_OLAP.validate_dimension procedure

Prerequisites

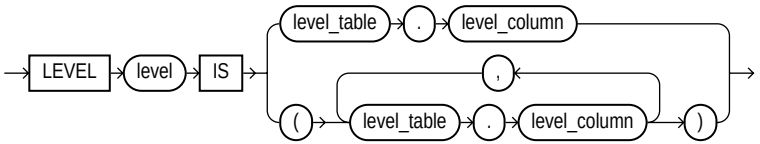
To create a dimension in your own schema, you must have the CREATE DIMENSION system privilege. To create a dimension in another user's schema, you must have the CREATE ANY DIMENSION system privilege. In either case, you must have the SELECT object privilege on any objects referenced in the dimension.

Syntax

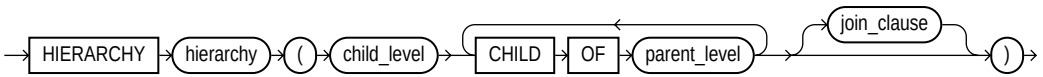
create_dimension::=



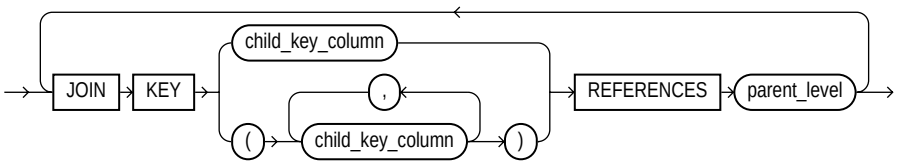
level_clause::=



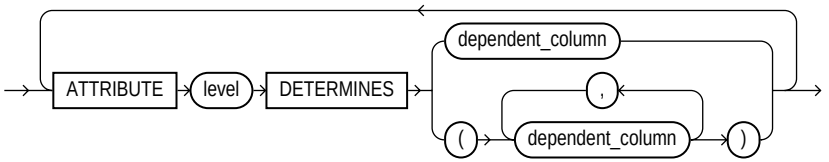
hierarchy_clause::=



join_clause::=



attribute_clause::=



Keywords and Parameters

schema

Specify the schema in which the dimension will be created. If you do not specify *schema*, Oracle creates the dimension in your own schema.

dimension

Specify the name of the dimension. The name must be unique within its schema.

level_clause

The *level_clause* defines a level in the dimension. A level defines dimension hierarchies and attributes.

level Specify the name of the level

level_table . level_column Specify the columns in the level. You can specify up to 32 columns. The tables you specify in this clause must already exist.

Restrictions on level columns:

- All of the columns in a level must come from the same table.
- If columns in different levels come from different tables, then you must specify the *join_clause*.
- The set of columns you specify must be unique to this level.
- The columns you specify cannot be specified in any other dimension.
- Each *level_column* must be non-null. (However, these columns need not have NOT NULL constraints.)

hierarchy_clause

The *hierarchy_clause* defines a linear hierarchy of levels in the dimension. Each hierarchy forms a chain of parent-child relationships among the levels in the dimension. Hierarchies in a dimension are independent of each other. They may (but need not) have columns in common.

Each level in the dimension should be specified at most once in this clause, and each level must already have been named in the *level_clause.*

hierarchy Specify the name of the hierarchy. This name must be unique in the dimension.

child_level Specify the name of a level that has an n:1 relationship with a parent level: the *level_columns* of *child_level* cannot be null, and each *child_level* value uniquely determines the value of the next named *parent_level*.

If the child *level_table* is different from the parent *level_table*, you must specify a join relationship between them in the *join_clause*.

parent_level Specify the name of a level.

join_clause

The *join_clause* lets you specify an inner equijoin relationship for a dimension whose columns are contained in multiple tables. This clause is required and permitted only when the columns specified in the hierarchy are not all in the same table.

child_key_column

Specify one or more columns that are join-compatible with columns in the parent level.

If you do not specify the schema and table of each *child_column*, the schema and table are inferred from the `CHILD OF` relationship in the *hierarchy_clause*. If you do specify the schema and column of a *child_key_column*, the schema and table must match the schema and table of columns in the child of *parent_level* in the *hierarchy_clause*.

parent_level

Specify the name of a level.

Restrictions on the *join_clause*:

- You can specify only one *join_clause* for a given pair of levels in the same hierarchy.
- The *child_key_columns* must be non-null and the parent key must be unique and non-null. You need not define constraints to enforce these conditions, but queries may return incorrect results if these conditions are not true.
- Each child key must join with a key in the *parent_level* table.
- Self-joins are not permitted. That is, the *child_key_columns* cannot be in the same table as *parent_level*.
- All of the child-key columns must come from the same table.

- The number of child-key columns must match the number of columns in *parent_level*, and the columns must be joinable.
- Do not specify multiple child key columns unless the parent level consists of multiple columns.

attribute_clause

The *attribute_clause* lets you specify the columns that are uniquely determined by a hierarchy level. The columns in *level* must all come from the same table as the *dependent_columns*. The *dependent_columns* need not have been specified in the *level_clause*.

For example, if the hierarchy levels are city, state, and country, then city might determine mayor, state might determine governor, and country might determine president.

Examples

Creating a Dimension: Example This statement was used to create the customers_dim dimension in the sample schema sh:

```
CREATE DIMENSION customers_dim
   LEVEL customer   IS (customers.cust_id)
   LEVEL city       IS (customers.cust_city)
   LEVEL state      IS (customers.cust_state_province)
   LEVEL country    IS (countries.country_id)
   LEVEL subregion  IS (countries.country_subregion)
   LEVEL region     IS (countries.country_region)
   HIERARCHY geog_rollup (
      customer      CHILD OF
      city          CHILD OF
      state         CHILD OF
      country       CHILD OF
      subregion     CHILD OF
      region
   JOIN KEY (customers.country_id) REFERENCES country
   )
   ATTRIBUTE customer DETERMINES
   (cust_first_name, cust_last_name, cust_gender,
    cust_marital_status, cust_year_of_birth,
    cust_income_level, cust_credit_limit)
   ATTRIBUTE country DETERMINES (countries.country_name)
;
```

CREATE DIRECTORY

Purpose

Use the CREATE DIRECTORY statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where external binary file LOBs (BFILEs) and external table data are located. You can use directory names when referring to BFILEs in your PL/SQL code and OCI calls, rather than hard coding the operating system path name, thereby providing greater file management flexibility.

All directories are created in a single namespace and are not owned by an individual's schema. You can secure access to the BFILEs stored within the directory structure by granting object privileges on the directories to specific users.

See Also:

- "Large Object (LOB) Datatypes" on page 2-27 for more information on BFILE objects
- GRANT on page 17-29 for more information on granting object privileges
- *external_table_clause* of CREATE TABLE on page 15-33

Prerequisites

You must have CREATE ANY DIRECTORY system privileges to create directories.

When you create a directory, you are automatically granted the READ and WRITE object privileges on the directory, and you can grant these privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

WRITE privileges on a directory are useful in connection with external tables. They let the grantee determine whether the external table agent can write a log file or a bad file to the directory.

You must also create a corresponding operating system directory for file storage. Your system or database administrator must ensure that the operating system directory has the correct read and write permissions for Oracle processes.

Privileges granted for the directory are created independently of the permissions defined for the operating system directory. Therefore, the two may or may not correspond exactly. For example, an error occurs if sample user hr is granted READ

privilege on the directory schema object but the corresponding operating system directory does not have READ permission defined for Oracle processes.

Syntax

create_directory::=

Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory.

Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges.

> **See Also:** DROP DIRECTORY on page 16-73 for information on removing a directory from the database

directory

Specify the name of the directory object to be created. The maximum length of *directory* is 30 bytes. You cannot qualify a directory object with a schema name.

> **Note:** Oracle does not verify that the directory you specify actually exists. Therefore, take care that you specify a valid directory in your operating system. In addition, if your operating system uses case-sensitive path names, be sure you specify the directory in the correct format. (However, you need not include a trailing slash at the end of the path name.)

'path_name'

Specify the full path name of the operating system directory on the server where the files are located. The single quotes are required, with the result that the path name is case sensitive.

Example

Creating a Directory: Examples The following statement creates a directory database object that points to a directory on the server:

```
CREATE DIRECTORY admin AS 'oracle/admin';
```

The following statement redefines directory database object `bfile_dir` to enable access to `BFILE`s stored in the operating system directory `/private1/lob/files`:

```
CREATE OR REPLACE DIRECTORY bfile_dir AS '/private1/LOB/files';
```

CREATE FUNCTION

Purpose

Use the CREATE FUNCTION statement to create a standalone stored function or a call specification. (You can also create a function as part of a package using the CREATE PACKAGE statement.)

A **stored function** (also called a **user function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

A **call specification** declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call specification tells Oracle which Java method, or which named function in which shared library, to invoke when a call is made. It also tells Oracle what type conversions to make for the arguments and return value.

See Also:

- CREATE PROCEDURE on page 14-62 for a general discussion of procedures and functions
- "Examples" on page 13-59 for examples of creating functions
- CREATE PACKAGE on page 14-50 for information on creating packages
- ALTER FUNCTION on page 9-61 for information on modifying a function
- CREATE LIBRARY on page 14-2 for information on shared libraries
- DROP FUNCTION on page 16-74 for information on dropping a standalone function
- *Oracle9i Application Developer's Guide - Fundamentals* for more information about registering external functions

Prerequisites

Before a stored function can be created, the user SYS must run a SQL script that is commonly called DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a function in your own schema, you must have the CREATE PROCEDURE system privilege. To create a function in another user's schema, you must have the CREATE ANY PROCEDURE system privilege. To replace a function in another user's schema, you must have the ALTER ANY PROCEDURE system privilege.

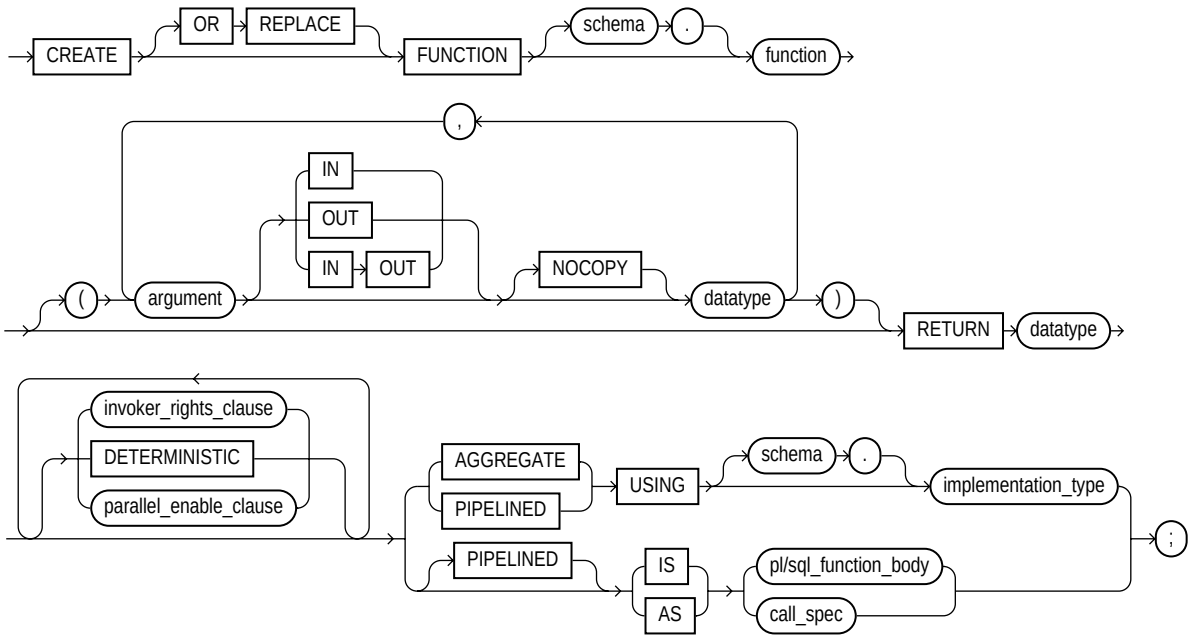
To invoke a call specification, you may need additional privileges (for example, EXECUTE privileges on C library for a C call specification).

To embed a CREATE FUNCTION statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

> **See Also:** *PL/SQL User's Guide and Reference* or *Oracle9i Java Stored Procedures Developer's Guide* for more information on such prerequisites

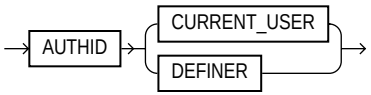
Syntax

create_function::=

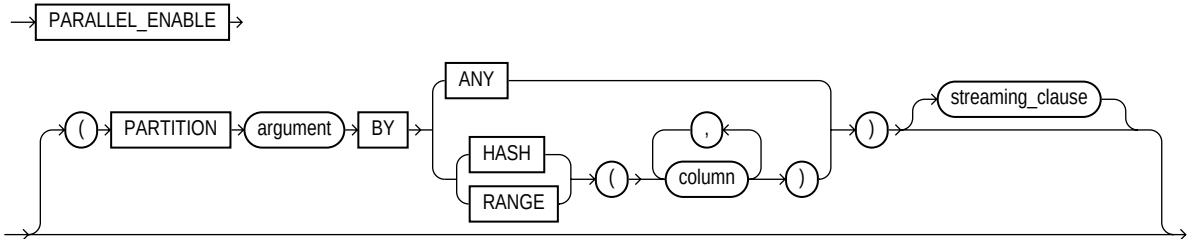


(*invoker_rights_clause::=* on page 13-51, *parallel_enable_clause::=* *on page 13-51)*

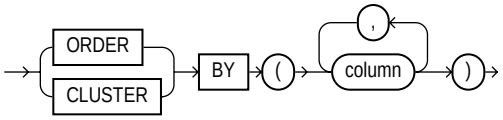
invoker_rights_clause::=



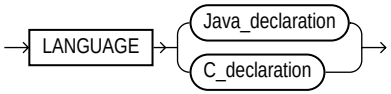
parallel_enable_clause::=



streaming_clause::=

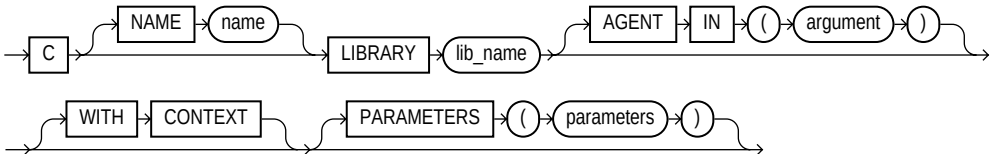


call_spec::=



Java_declaration::=

C_declaration::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the function if it already exists. Use this clause to change the definition of an existing function without dropping, re-creating, and regranting object privileges previously granted on the function. If you redefine a function, Oracle recompiles it.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, Oracle marks the indexes DISABLED.

> **See Also:** ALTER FUNCTION on page 9-61 for information on recompiling functions

schema

Specify the schema to contain the function. If you omit *schema*, Oracle creates the function in your current schema.

function

Specify the name of the function to be created. If creating the function results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SHOW ERRORS command.

Restrictions on user-defined functions:

User-defined functions cannot be used in situations that require an unchanging definition. Thus, you cannot use user-defined functions:

- In a CHECK constraint clause of a CREATE TABLE or ALTER TABLE statement
- In a DEFAULT clause of a CREATE TABLE or ALTER TABLE statement

In addition, when a function is called from within a query or DML statement, the function cannot:

- Have OUT or IN OUT parameters
- Commit or roll back the current transaction, create a savepoint or roll back to a savepoint, or alter the session or the system. DDL statements implicitly commit the current transaction, so a user-defined function cannot execute any DDL statements.
- Write to the database, if the function is being called from a SELECT statement. However, a function called from a subquery in a DML statement can write to the database.
- Write to the same table that is being modified by the statement from which the function is called, if the function is called from a DML statement.

Except for the restriction on OUT and IN OUT parameters, Oracle enforces these restrictions not only for the function called directly from the SQL statement, but also for any functions that function calls, and on any functions called from the SQL statements executed by that function or any function it calls.

See Also: "Creating a Function: Examples" on page 13-59

argument

Specify the name of an argument to the function. If the function does not accept arguments, you can omit the parentheses following the function name.

Restriction on function arguments: If you are creating an aggregate function, you can specify only one argument.

IN Specify IN to indicate that you must supply a value for the argument when calling the function. This is the default.

OUT Specify OUT to indicate that the function will set the value of the argument.

IN OUT Specify IN OUT to indicate that a value for the argument can be supplied by you and may be set by the function.

NOCOPY Specify NOCOPY to instruct Oracle to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an index-by table, or a varray to an OUT or IN OUT parameter. (IN parameter values are always passed NOCOPY.)

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter (or assignments made to this parameter may show immediately in a package variable) if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the procedure is exited with an unhandled exception, any assignment made to this parameter may be visible in the caller's variable.

These effects may or may not occur on any particular call. You should use NOCOPY only when these effects would not matter.

RETURN Clause

For datatype, specify the datatype of the function's return value. Because every function must return a value, this clause is required. The return value can have any datatype supported by PL/SQL.

Note: Oracle SQL does not support calling of functions with boolean parameters or returns. Therefore, if your user-defined functions will be called from SQL statements, you must design them to return numbers (0 or 1) or character strings ('TRUE' or 'FALSE').

The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of the return value from the environment from which the function is called.

If the return type is `SYS.AnyDataSet` and you intend to use the function in the `FROM` clause of a query, then you must also specify the `PIPELINED` clause and define a describe method (`ODCITableDescribe`) as part of the implementation type of the function.

See Also:

- *PL/SQL User's Guide and Reference* for information on PL/SQL datatypes
- *Oracle9i Data Cartridge Developer's Guide* for information on defining the `ODCITableDescribe` function

invoker_rights_clause

The *`invoker_rights_clause`* lets you specify whether the function executes with the privileges and in the schema of the user who owns it or with the privileges and in the schema of `CURRENT_USER`.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the function.

AUTHID Clause

- Specify `CURRENT_USER` if you want the function to execute with the privileges of `CURRENT_USER`. This clause creates an **invoker-rights function**.

  This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of `CURRENT_USER`. External names in all other statements resolve in the schema in which the function resides.

- Specify `DEFINER` if you want the function to execute with the privileges of the owner of the schema in which the function resides, and that external names resolve in the schema where the function resides. This is the default and creates a **defined-rights function**.

See Also:

- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *PL/SQL User's Guide and Reference*

DETERMINISTIC Clause

Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its arguments.

You must specify this keyword if you intend to call the function in the expression of a function-based index or from the query of a materialized view that is marked REFRESH FAST or ENABLE QUERY REWRITE. When Oracle encounters a deterministic function in one of these contexts, it attempts to use previously calculated results when possible rather than re-executing the function.

Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the function's return result. The results of doing so will not be captured if Oracle chooses not to reexecute the function.

The following semantic rules govern the use of the DETERMINISTIC clause:

- You can declare a top-level subprogram DETERMINISTIC.
- You can declare a package-level subprogram DETERMINISTIC in the package specification, but not in the package body.
- You cannot declare DETERMINISTIC a private subprogram (declared inside another subprogram or inside a package body).
- A DETERMINISTIC subprogram can call another subprogram whether the called program is declared DETERMINISTIC or not.

See Also:

- *Oracle9i Data Warehousing Guide* for information on materialized views
- CREATE INDEX on page 13-62 for information on function-based indexes

parallel_enable_clause

PARALLEL_ENABLE is an optimization hint indicating that the function can be executed from a parallel execution server of a parallel query operation. The function should not use session state, such as package variables, as those variables may not be shared among the parallel execution servers.

- The optional PARTITION *argument* BY clause is used only with functions that have a REF CURSOR argument type. It lets you define the partitioning of the inputs to the function from the REF CURSOR argument.

  Partitioning the inputs to the function affects the way the query is parallelized when the function is used as a table function (that is, in the FROM clause of the query). ANY indicates that the data can be partitioned randomly among the parallel execution servers. Alternatively, you can specify RANGE or HASH partitioning on a specified column list.

- The optional *streaming_clause* lets you order or cluster the parallel processing by a specified column list.

  - ORDER BY indicates that the rows on a parallel execution server must be locally ordered.
  - CLUSTER BY indicates that the rows on a parallel execution server must have the same key values as specified by the *column_list*.

The columns specified in all of these optional clauses refer to columns that are returned by the REF CURSOR argument of the function.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals, Oracle9i Data Cartridge Developer's Guide,* and *PL/SQL User's Guide and Reference* for more information on user-defined aggregate functions

PIPELINED Clause

Use PIPELINED to instruct Oracle to return the results of a **table function** iteratively. A table function returns a collection type (a nested table or varray). You query table functions by using the TABLE keyword before the function name in the FROM clause of the query. For example:

```
SELECT * FROM TABLE(function_name(...))
```

Oracle then returns rows as they are produced by the function.

- If you specify the keyword PIPELINED alone (PIPELINED IS ...), the PL/SQL function body should use the PIPE keyword. This keyword instructs Oracle to

return single elements of the collection out of the function, instead of returning the whole collection as a single value.

- You can specify PIPELINED USING *implementation_type* clause if you want to predefine an interface containing the start, fetch, and close operations. The implementation type must implement the ODCITable interface, and must exist at the time the table function is created. This clause is useful for table functions that will be implemented in external languages such as C++ and Java.

  If the return type of the function is SYS.AnyDataSet, then you must also define a describe method (ODCITableDescribe) as part of the implementation type of the function.

  See Also:

  - *PL/SQL User's Guide and Reference* and *Oracle9i Application Developer's Guide - Fundamentals* for more information on table functions
  - *Oracle9i Data Cartridge Developer's Guide* for information on ODCI routines

AGGREGATE USING Clause

Specify AGGREGATE USING to identify this function as an **aggregate function**, or one that evaluates a group of rows and returns a single row. You can specify aggregate functions in the SELECT list, HAVING clause, and ORDER BY clause.

Note: When you specify a user-defined aggregate function in a query, you can treat it as an **analytic function** (one that operates on a query result set). To do so, use the OVER *analytic_clause* syntax available for built-in analytic functions. See "Analytic Functions" on page 6-9 for syntax and semantics.

In the USING clause, specify the name of the implementation type of the function. The implementation type must be an object type containing the implementation of the ODCIAggregate routines. If you do not specify *schema*, Oracle assumes that the implementation type is in your own schema.

Restriction on the AGGREGATE USING clause: If you specify this clause, you can specify only one input argument for the function.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for information on ODCI routines and "Creating Aggregate Functions: Example" on page 13-60

IS | AS Clause

pl/sql_subprogram_body Use the *pl/sql_subprogram_body* to declare the function in a PL/SQL subprogram body.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information on PL/SQL subprograms and "Using a Packaged Procedure in a Function: Example" on page 13-61

call_spec Use the *call_spec* to map a Java or C method name, parameter types, and return type to their SQL counterparts. In *Java_declaration, 'string'* identifies the Java implementation of the method.

> **See Also:**
>
> - *Oracle9i Java Stored Procedures Developer's Guide*
> - *Oracle9i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

AS EXTERNAL AS EXTERNAL is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

Examples

Creating a Function: Examples The following statement creates the function get_bal on the sample table oe.orders (the PL/SQL is in italics):

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
   RETURN NUMBER
   IS acc_bal NUMBER(11,2);
   BEGIN
      SELECT order_total
      INTO acc_bal
      FROM orders
      WHERE customer_id = acc_no;
      RETURN(acc_bal);
```

```
    END;
/
```

The get_bal function returns the balance of a specified account.

When you call the function, you must specify the argument acc_no, the number of the account whose balance is sought. The datatype of acc_no is NUMBER.

The function returns the account balance. The RETURN clause of the CREATE FUNCTION statement specifies the datatype of the return value to be NUMBER.

The function uses a SELECT statement to select the balance column from the row identified by the argument acc_no in the orders table. The function uses a RETURN statement to return this value to the environment in which the function is called.

The function created in the preceding example can be used in a SQL statement. For example:

```
SELECT get_bal(165) FROM DUAL;

GET_BAL(165)
------------
        2519
```

The following statement creates PL/SQL standalone function get_val that registers the C routine c_get_val as an external function. (The parameters have been omitted from this example; the PL/SQL is in italics.)

```
CREATE FUNCTION get_val
   ( x_val IN NUMBER,
    y_val IN NUMBER,
    image IN LONG RAW )
   RETURN BINARY_INTEGER AS LANGUAGE C
      NAME "c_get_val"
      LIBRARY c_utils
      PARAMETERS (...);
```

Creating Aggregate Functions: Example The next statement creates an aggregate function called SecondMax to aggregate over number values. It assumes that the object type SecondMaxImpl routines contains the implementations of the ODCIAggregate routines:

```
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER
    PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for the complete implementation of type and type body for `SecondMaxImpl`

You would use such an aggregate function in a query like the following statement, which queries the sample table `hr.employees`:

```
SELECT SecondMax(salary), department_id
      FROM employees
      GROUP BY department_id
      HAVING SecondMax(salary) > 9000;

SECONDMAX(SALARY) DEPARTMENT_ID
----------------- -------------
            13500            80
            17000            90
```

Using a Packaged Procedure in a Function: Example The following statement creates a function that uses a `DBMS_LOB` procedure to return the length of a `CLOB` column:

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
   RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

> **See Also:** "Creating a Function-Based Index on a LOB Column: Example" on page 13-85 to see how to use this function to create a function-based index

CREATE INDEX

Purpose

Use the CREATE INDEX statement to create an index on

- One or more columns of a table, a partitioned table, an index-organized table, or a cluster
- One or more scalar typed object attributes of a table or a cluster
- A nested table storage table for indexing a nested table column

An **index** is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. Oracle supports several types of index:

- Normal indexes (by default, Oracle creates B-tree indexes)
- **Bitmap indexes**, which store rowids associated with a key value as a bitmap
- **Partitioned indexes**, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table
- **Function-based indexes**, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include functions (built-in or user-defined).
- **Domain indexes**, which are instances of an application-specific index of type *indextype*

See Also:

- *Oracle9i Database Concepts* for a discussion of indexes
- ALTER INDEX on page 9-64
- DROP INDEX on page 16-76

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

- The table or cluster to be indexed must be in your own schema.
- You must have INDEX object privilege on the table to be indexed.

- You must have CREATE ANY INDEX system privilege.

To create an index in another schema, you must have CREATE ANY INDEX system privilege. Also, the owner of the schema to contain the index must have either the UNLIMITED TABLESPACE system privilege or space quota on the tablespaces to contain the index or index partitions.

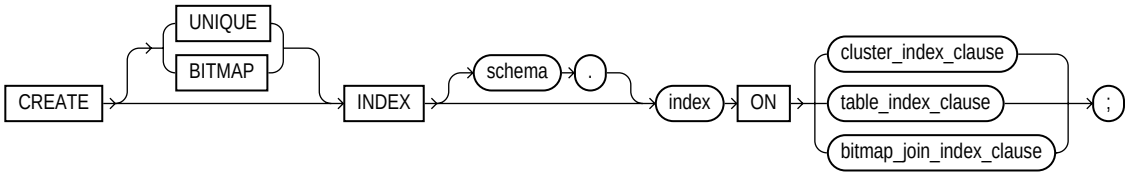
To create a domain index in your own schema, in addition to the prerequisites for creating a conventional index, you must also have EXECUTE privilege on the indextype. If you are creating a domain index in another user's schema, the index owner also must have EXECUTE privilege on the indextype and its underlying implementation type. Before creating a domain index, you should first define the indextype.

To create a function-based index in your own schema on your own table, in addition to the prerequisites for creating a conventional index, you must have the QUERY REWRITE system privilege. To create the index in another schema or on another schema's table, you must have the GLOBAL QUERY REWRITE privilege. In both cases, the table owner must also have the EXECUTE object privilege on the function(s) used in the function-based index. In addition, in order for Oracle to use function-based indexes in queries, the QUERY_REWRITE_ENABLED parameter must be set to TRUE, and the QUERY_REWRITE_INTEGRITY parameter must be set to TRUSTED.

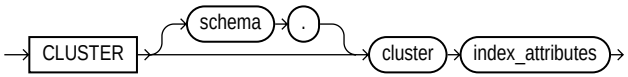
See Also: CREATE INDEXTYPE on page 13-91

Syntax

create_index::=

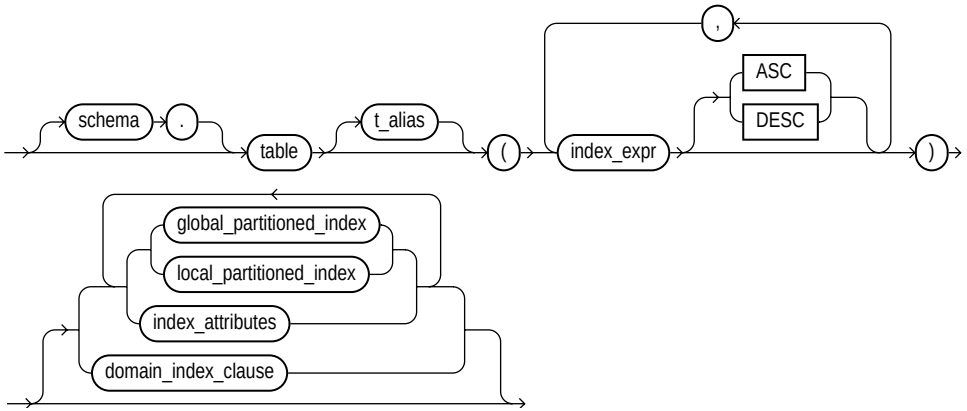


cluster_index_clause::=



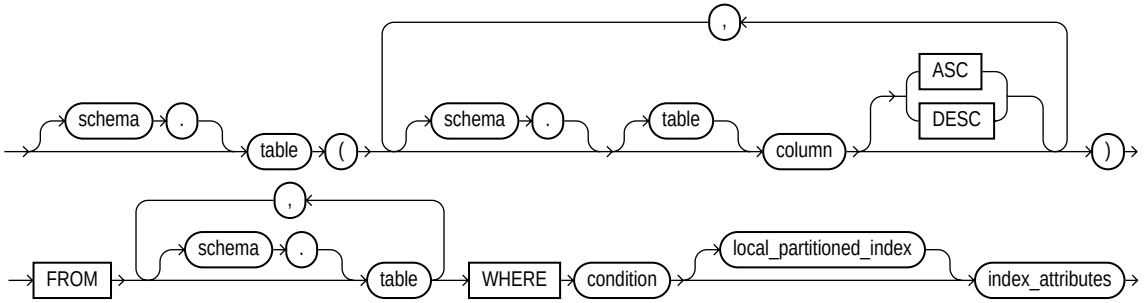
(*index_attributes::=* on page 13-65)

table_index_clause::=



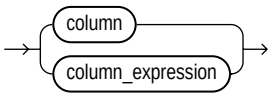
(*global_partitioned_index::=* on page 13-66, *local_partitioned_index::=* on page 13-66, *index_attributes::=* on page 13-65, *domain_index_clause::=* on page 13-66)

bitmap_join_index_clause::=

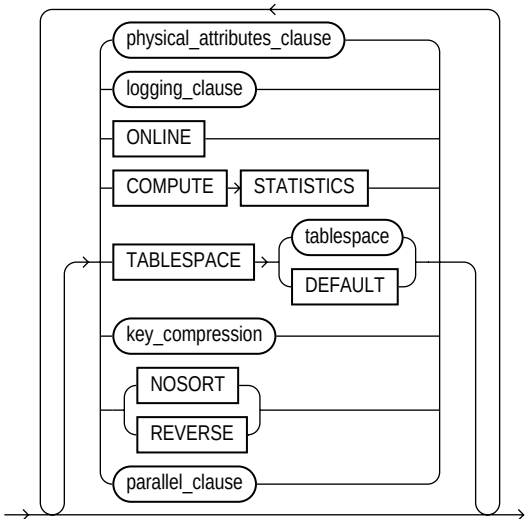


(*local_partitioned_index::=* on page 13-66, *index_attributes::=* on page 13-65)

index_expr::=

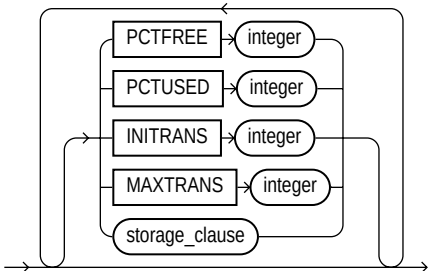


index_attributes::=



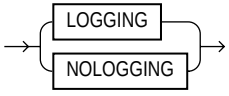
(*physical_attributes_clause::=* on page 13-65, *logging_clause::=* on page 13-65, *key_compression::=* on page 13-66, *parallel_clause::=* on page 13-68)

physical_attributes_clause::=

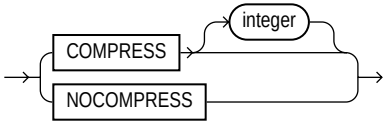


(storage_clause::= on page 7-58)

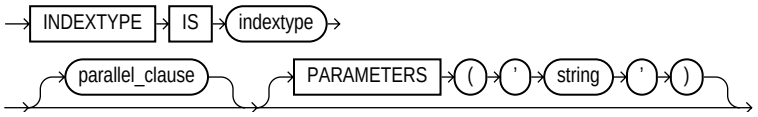
logging_clause::=



key_compression::=

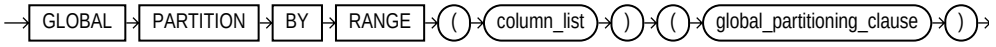


domain_index_clause::=



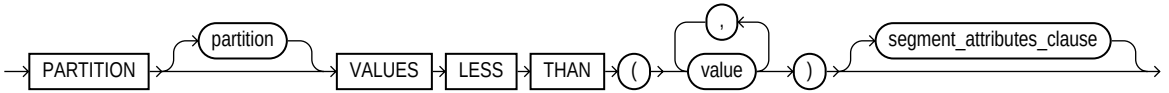
(*parallel_clause::=* on page 13-68)

global_partitioned_index::=



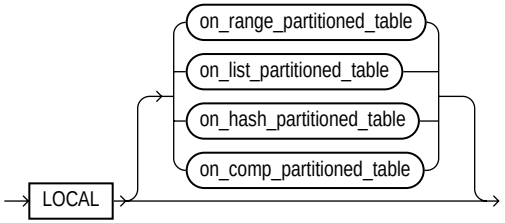
(*global_partitioning_clause::=* on page 13-66)

global_partitioning_clause::=



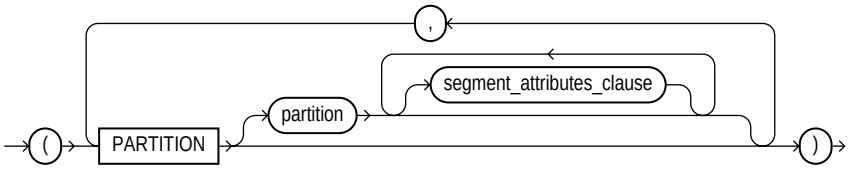
(*segment_attributes_clause::=* on page 13-67)

local_partitioned_index::=



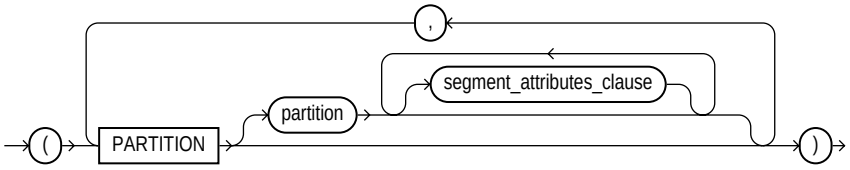
(*on_range_partitioned_table::=* on page 13-67, *on_list_partitioned_table::=* on page 13-67, *on_hash_partitioned_table::=* on page 13-67, *on_comp_partitioned_table::=* on page 13-68)

on_range_partitioned_table::=



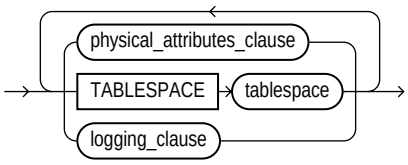
(*segment_attributes_clause::=* on page 13-67)

on_list_partitioned_table::=



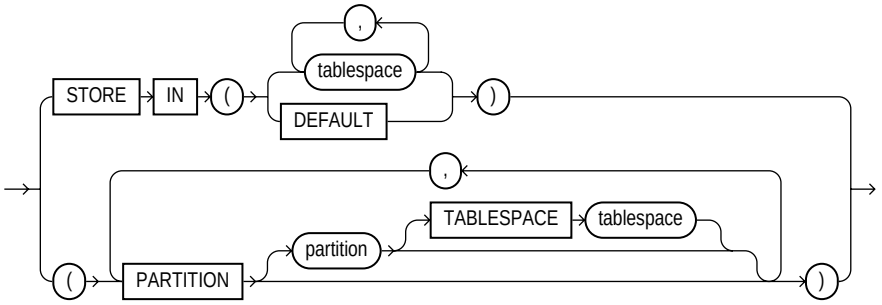
(*segment_attributes_clause::=* on page 13-67)

segment_attributes_clause::=

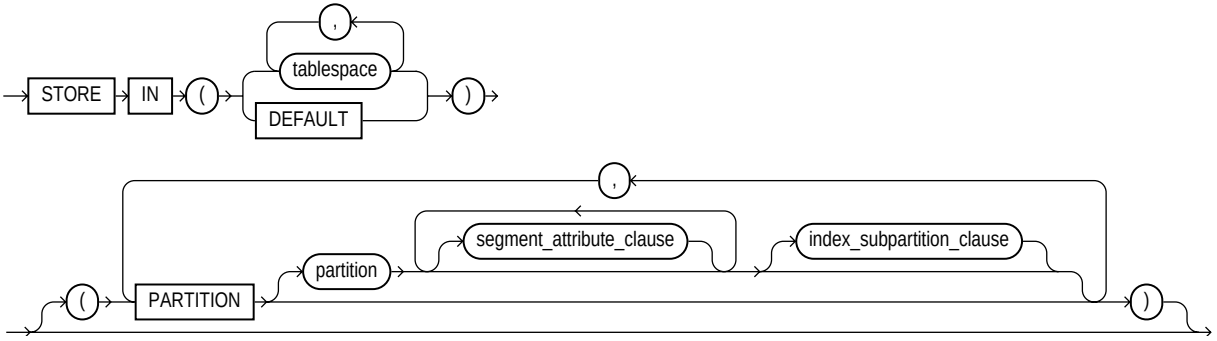


(*physical_attributes_clause::=* on page 13-65, *logging_clause::=* on page 13-65

on_hash_partitioned_table::=

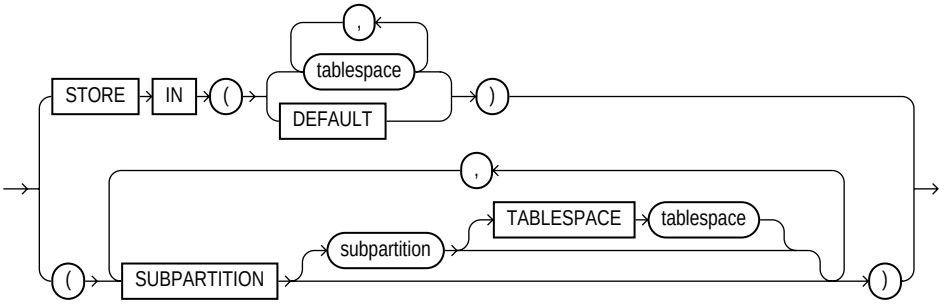


on_comp_partitioned_table::=

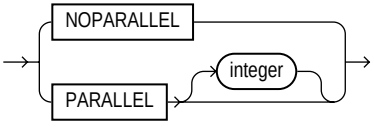


(*segment_attributes_clause::=* on page 13-67, *index_subpartition_clause::=* on page 13-68)

index_subpartition_clause::=



parallel_clause::=



(*storage_clause* on page 7-56)

Keywords and Parameters

UNIQUE

Specify UNIQUE to indicate that the value of the column (or columns) upon which the index is based must be unique. If the index is local nonprefixed (see *local_partitioned_index*), then the index key must contain the partitioning key.

Restrictions on unique indexes:

- You cannot specify both UNIQUE and BITMAP.
- You cannot specify UNIQUE for a domain index.

> **See Also:** *constraints* on page 7-5 for information on integrity constraints

BITMAP

Specify BITMAP to indicate that *index* is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. The internal representation of bitmaps is best suited for applications with low levels of concurrent transactions, such as data warehousing.

> **Note:** Oracle does not index table rows in which all key columns are null except in the case of bitmap indexes. Therefore, if you want an index on all rows of a table, you must either specify NOT NULL constraints for the index key columns or create a bitmap index.

Restrictions on bitmapped indexes:

- You cannot specify BITMAP when creating a global partitioned index.
- You cannot create a bitmapped secondary index on an index-organized table unless the index-organized table has a mapping table associated with it.
- You cannot specify both UNIQUE and BITMAP.
- You cannot specify BITMAP for a domain index.

See Also:

- *Oracle9i Database Concepts* and *Oracle9i Database Performance Tuning Guide and Reference* for more information about using bitmap indexes
- CREATE TABLE on page 15-7 for information on mapping tables
- "Bitmap Index Example" on page 13-88

schema

Specify the schema to contain the index. If you omit *schema*, Oracle creates the index in your own schema.

index

Specify the name of the index to be created.

See Also: "Creating an Index: Example" on page 13-83 and "Create an Index on an XMLType Table: Example" on page 13-84

cluster_index_clause

Use the *cluster_index_clause* to identify the cluster for which a cluster index is to be created. If you do not qualify cluster with *schema*, Oracle assumes the cluster is in your current schema. You cannot create a cluster index for a hash cluster.

See Also: CREATE CLUSTER on page 13-2 and "Creating a Cluster Index: Example" on page 13-84

table_index_clause

Specify the table (and its attributes) on which you are defining the index. If you do not qualify *table* with *schema*, Oracle assumes the table is contained in your own schema.

You create an index on a nested table column by creating the index on the nested table storage table. Include the NESTED_TABLE_ID pseudocolumn of the storage table to create a UNIQUE index, which effectively ensures that the rows of a nested table value are distinct.

See Also: "Indexes on Nested Tables: Example" on page 13-89

Restrictions on the *table_index_clause*:

- If the index is locally partitioned, then *table* must be partitioned.
- If the table is index-organized, this statement creates a secondary index. You cannot specify REVERSE for this secondary index, and the combined size of the index key and the logical rowid should be less than half the block size.
- If *table* is a temporary table, the index will also be temporary with the same scope (session or transaction) as *table*. The following restrictions apply to indexes on temporary table:
  - The index cannot be a partitioned index or a domain index.
  - You cannot specify the *physical_attributes_clause* or the *parallel_clause*.
  - You cannot specify LOGGING, NOLOGGING, or TABLESPACE.

  See Also: CREATE TABLE on page 15-7 and *Oracle9i Database Concepts* for more information on temporary tables

t_alias

Specify a correlation name (alias) for the table upon which you are building the index.

Note: This alias is required if the *index_expr* references any object type attributes or object type methods. See "Creating a Function-based Index on a Type Method: Example" on page 13-86 and "Indexing on Substitutable Columns: Examples" on page 13-89.

---

index_expr

For *index_expr*, specify the column or column expression upon which the index is based.

column Specify the name of a column in the table. A bitmap index can have a maximum of 30 columns. Other indexes can have as many as 32 columns.

You can create an index on a scalar object attribute column or on the system-defined NESTED_TABLE_ID column of the nested table storage table. If you specify an object attribute column, the column name must be qualified with the table name. If you specify a nested table column attribute, it must be qualified with the outermost

table name, the containing column name, and all intermediate attribute names leading to the nested table column attribute.

Restriction: You cannot create an index on columns or attributes whose type is user-defined, LONG, LONG RAW, LOB, or REF, except that Oracle supports an index on REF type columns or attributes that have been defined with a SCOPE clause.

column_expression Specify an expression built from columns of *table*, constants, SQL functions, and user-defined functions. When you specify *column_expression*, you create a **function-based index**.

> **See Also:** "Notes on Function-Based Indexes:" on page 13-72, "Restrictions on function-based indexes:" on page 13-73, and "Function-Based Index Examples" on page 13-85

Name resolution of the function is based on the schema of the index creator. User-defined functions used in *column_expression* are fully name resolved during the CREATE INDEX operation.

After creating a function-based index, collect statistics on both the index and its base table using the ANALYZE statement. Oracle cannot use the function-based index until these statistics have been generated.

> **See Also:** ANALYZE on page 12-33

Notes on Function-Based Indexes:

When you subsequently query a table that uses a function-based index, you must ensure in the query that *column_expression* is not null. However, Oracle will use a function-based index in a query even if the columns specified in the WHERE clause are in a different order than their order in the *column_expression* that defined the function-based index.

> **See Also:** "Function-Based Index Examples" on page 13-85

If the function on which the index is based becomes invalid or is dropped, Oracle marks the index DISABLED. Queries on a DISABLED index fail if the optimizer chooses to use the index. DML operations on a DISABLED index fail unless the index is also marked UNUSABLE **and** the parameter SKIP_UNUSABLE_INDEXES is set to true.

> **See Also:** ALTER SESSION on page 10-2 for more information on this parameter

Oracle's use of function-based indexes is also affected by the setting of the QUERY_REWRITE_ENABLED session parameter.

> **See Also:** ALTER SESSION on page 10-2

If a public synonym for a function, package, or type is used in *column_expression*, and later an actual object with the same name is created in the table owner's schema, then Oracle will disable the function-based index. When you subsequently enable the function-based index using ALTER INDEX ... ENABLE or ALTER INDEX ... REBUILD, the function, package, or type used in the *column_expression* will continue to resolve to the function, package, or type to which the public synonym originally pointed. It will not resolve to the new function, package, or type.

If the definition of a function-based index generates internal conversion to character data, use caution when changing NLS parameter settings. Function-based indexes use the current database settings for NLS parameters. If you reset these parameters at the session level, queries using the function-based index may return incorrect results. Two exceptions are the collation parameters (NLS_SORT and NLS_COMP). Oracle handles the conversions correctly even if these have been reset at the session level.

Restrictions on function-based indexes:

- Any user-defined function referenced in *column_expression* must be DETERMINISTIC.
- For a function-based globally partitioned index, the *column_expression* cannot be the partitioning key.
- *column_expression* can be any form of expression except a scalar subquery expression
- All functions must be specified with parentheses, even if they have no parameters. Otherwise Oracle interprets them as column names.
- Any function you specify in *column_expression* must return a repeatable value. For example, you cannot specify the SYSDATE or USER function or the ROWNUM pseudocolumn.
- The *column_expression* cannot contain any aggregate functions.
- You cannot create a function-based index on a nested table.

> **See Also:** CREATE FUNCTION on page 13-49 and *PL/SQL User's Guide and Reference*

ASC | DESC

Use ASC or DESC to indicate whether the index should be created in ascending or descending order. Indexes on character data are created in ascending or descending order of the character values in the database character set.

Oracle treats descending indexes as if they were function-based indexes. You do not need the QUERY REWRITE or GLOBAL QUERY REWRITE privileges to create them, as you do with other function-based indexes. However, as with other function-based indexes, Oracle does not use descending indexes until you first analyze the index and the table on which the index is defined. See the *column_expression* clause of this statement.

Restriction on ASC and DESC: You cannot specify either of these clauses for a domain index. You cannot specify DESC for a reverse index. Oracle ignores DESC if *index* is bitmapped or if the COMPATIBLE initialization parameter is set to a value less than 8.1.0.

index_attributes

physical_attributes_clause Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the index.

If you omit this clause, Oracle uses the following default values:

- PCTFREE: 10
- INITRANS: 2
- MAXTRANS: Depends on data block size

Restriction on the *physical_attributes_clause***:** You cannot specify the PCTUSED parameter for an index.

See Also:

- *physical_attributes_clause* on page 7-52 for a complete description of the parameters of this clause
- *storage_clause* on page 7-56 for a complete description of storage parameters, including default values

TABLESPACE For *tablespace*, specify the name of the tablespace to hold the index, index partition, or index subpartition. If you omit this clause, Oracle creates the index in the default tablespace of the owner of the schema containing the index.

For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions or subpartitions added to the local index will be created in the same tablespace(s) as the corresponding partitions or subpartitions of the underlying table.

COMPRESS Specify COMPRESS to enable key compression, which eliminates repeated occurrence of key column values and may substantially reduce storage. Use *integer* to specify the prefix length (number of prefix columns to compress).

- For unique indexes, the valid range of prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the valid range of prefix length values is from 1 to the number of key columns. The default prefix length is the number of key columns.

  Oracle compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns.

  Restriction on key compression: You cannot specify COMPRESS for a bitmap index.

  See Also: "Compressing an Index: Example" on page 13-84

NOCOMPRESS Specify NOCOMPRESS to disable key compression. This is the default.

NOSORT Specify NOSORT to indicate to Oracle that the rows are stored in the database in ascending order, so that Oracle does not have to sort the rows when creating the index. If the rows of the indexed column or columns are not stored in ascending order, Oracle returns an error. For greatest savings of sort time and space, use this clause immediately after the initial load of rows into a table.

Restrictions on NOSORT:

- You cannot specify REVERSE with this clause.
- You cannot use this clause to create a cluster, partitioned, or bitmap index.
- You cannot specify this clause for a secondary index on an index-organized table.

REVERSE Specify REVERSE to store the bytes of the index block in reverse order, excluding the rowid.

Restrictions on REVERSE:

- You cannot specify NOSORT with this clause.
- You cannot reverse a bitmap index or an index-organized table.

logging_clause Specify whether the creation of the index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. This setting also determines whether subsequent Direct Loader (SQL*Loader) and direct-path INSERT operations against the index are logged or not logged. LOGGING is the default.

If *index* is nonpartitioned, this clause specifies the logging attribute of the index.

If *index* is partitioned, this clause determines:

- The default value of all partitions specified in the CREATE statement (unless you specify the *logging_clause* in the PARTITION description clause)
- The default value for the segments associated with the index partitions
- The default value for local index partitions or subpartitions added implicitly during subsequent ALTER TABLE ... ADD PARTITION operations

The logging attribute of the index is independent of that of its base table.

If you omit this clause, the logging attribute is that of the tablespace in which it resides.

See Also:

- *logging_clause* on page 7-45 for a full description of this clause
- *Oracle9i Database Concepts* and *Oracle9i Data Warehousing Guide* for more information about logging and parallel DML
- "Creating an Index in NOLOGGING Mode: Example" on page 13-84

ONLINE Specify ONLINE to indicate that DML operations on the table will be allowed during creation of the index.

Restrictions on online index building:

- Parallel DML is not supported during online index building. If you specify ONLINE and then issue parallel DML statements, Oracle returns an error.
- You cannot specify ONLINE for a bitmap index or a cluster index.

- You cannot specify `ONLINE` for a conventional index on a `UROWID` column.
- For a unique index on an index-organized table, the number of index key columns plus the number of primary key columns in the index-organized table cannot exceed 32.

> **See Also:** *Oracle9i Database Concepts* for a description of online index building and rebuilding

COMPUTE STATISTICS Specify `COMPUTE STATISTICS` to collect statistics at relatively little cost during the creation of an index. These statistics are stored in the data dictionary for ongoing use by the optimizer in choosing a plan of execution for SQL statements.

The types of statistics collected depend on the type of index you are creating.

> **Note:** If you create an index using another index (instead of a table), the original index might not provide adequate statistical information. Therefore, Oracle generally uses the base table to compute the statistics, which will improve the statistics but may negatively affect performance.

Additional methods of collecting statistics are available in PL/SQL packages and procedures.

> **See Also:** *Oracle9i Supplied PL/SQL Packages and Types Reference* and "Computing Index Statistics: Example" on page 13-84

parallel_clause

Specify the *`parallel_clause`* if you want creation of the index to be parallelized.

> **Note:** The syntax of the *`parallel_clause`* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify `NOPARALLEL` for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Index Partitioning Clauses

Use the *global_partitioned_index* clause and the *local_partitioned_index* clauses to partition *index*.

Note: The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle9i Database Administrator's Guide* for a discussion of these restrictions.

See Also: "Partitioned Index Examples" on page 13-87

global_partitioned_index The *global_partitioned_index* clause lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes. Oracle will partition the global index on the ranges of values from the table columns you specify in *column_list*. You cannot specify this clause for a local index.

The *column_list* must specify a left prefix of the index column list. That is, if the index is defined on columns a, b, and c, then for *column_list* you can specify (a, b, c), or (a, b), or (a, c), but you cannot specify (b, c) or (c) or (b, a).

Restrictions on *column_list*:

- You cannot specify more than 32 columns in *column_list*.
- The columns cannot contain the ROWID pseudocolumn or a column of type ROWID.

Note: If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

See Also: *Oracle9i Database Globalization Support Guide* for more information on character set support

index_partitioning_clause Use this clause to describe the individual index partitions. The number of repetitions of this clause determines the number of partitions. If you omit *partition*, Oracle generates a name with the form SYS_P*n*.

For VALUES LESS THAN (*value_list*), specify the (noninclusive) upper bound for the current partition in a global index. The value list is a comma-delimited, ordered list of literal values corresponding to the column list in the *global_partitioned_index* clause. Always specify MAXVALUE as the value of the last partition.

Note: If the index is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, you must use the TO_DATE function with a 4-character format mask for the year. The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT.

See Also:

- *Oracle9i Database Globalization Support Guide* for more information on these initialization parameters
- "Range Partitioning Example" on page 15-73

local_partitioned_index

The *local_partitioned_index* clauses let you specify that the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as *table*. Oracle automatically maintains LOCAL index partitioning as the underlying table is repartitioned.

on_range_partitioned_table Specify the name and attributes of an index on a range-partitioned table.

- For PARTITION, specify the names of the individual partitions. The number of clauses determines the number of partitions. For a local index, the number of index partitions must be equal to the number of the table partitions, and in the same order.
- If you omit *partition*, Oracle generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, the form SYS_P*n* is used.

on_list_partitioned_table The *on_list_partitioned_table* clause is identical to *on_range_partitioned_table* on page 13-79.

on_hash_partitioned_table Specify the name and attributes of an index on a hash-partitioned table. If you do not specify *partition*, Oracle uses the name of the corresponding base table partition, unless it conflicts with an explicitly specified name of another index partition. In this case, Oracle generates a name of the form SYS_P*nnn*.

You can optionally specify TABLESPACE for all index partitions or for one or more individual partitions. If you do not specify TABLESPACE at the index or partition level, Oracle stores each index partition in the same tablespace as the corresponding table partition.

on_comp_partitioned_table Specify the name and attributes of an index on a composite-partitioned table. The first STORE IN clause specifies the default tablespace for the index subpartitions. You can override this storage by specifying a different tablespace in the *index_subpartitioning_clause*.

If you do not specify TABLESPACE for subpartitions either in this clause or in the *index_subpartitioning_clause*, Oracle uses the tablespace specified for *index*. If you also do not specify TABLESPACE for *index*, Oracle stores the subpartition in the same tablespace as the corresponding table subpartition.

STORE IN The STORE IN clause lets you specify how index hash partitions (for a hash-partitioned index) or index subpartitions (for a composite-partitioned index) are to be distributed across various tablespaces. The number of tablespaces does not have to equal the number of index partitions. If the number of index partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

- The DEFAULT clause is valid only for a local index on a hash or composite-partitioned table. This clause overrides any tablespace specified at the index level for a partition or subpartition, and stores the index partition or

subpartition in the same partition as the corresponding table partition or subpartition.

- The *index_subpartition_clause* lets you specify one or more tablespaces in which to store all subpartitions in *partition* or one or more individual subpartitions in *partition*. The subpartition inherits all other attributes from *partition*. Attributes not specified for *partition* are inherited from *index*.

domain_index_clause

Use the *domain_index_clause* to indicate that *index* is a domain index, which is an instance of an application-specific index of type *indextype*.

Note: Creating a domain index requires a number of preceding operations. You must first create an implementation type for an indextype. You must also create a functional implementation and then create an operator that uses the function. Next you create an indextype, which associates the implementation type with the operator. Finally, you create the domain index using this clause.

Appendix D, "Examples", contains an example of a simple domain index, including all of these operations. The examples are collected in one appendix because they would be difficult to follow if scattered throughout this reference under their appropriate SQL statements.

---

column Specify the table columns or object attributes on which the index is defined. You can define multiple domain indexes on a single column only if the underlying indextypes are different and the indextypes support a disjoint set of user-defined operators.

Restriction on *column*: You cannot create a domain index on a column of datatype REF, varray, nested table, LONG, or LONG RAW.

indextype For *indextype*, specify the name of the indextype. This name should be a valid schema object that you have already defined.

Note: If you have installed Oracle Text, you can use various built-in indextypes to create Oracle Text domain indexes. For more information on Oracle Text and the indexes it uses, please refer to *Oracle Text Reference*.

See Also: CREATE INDEXTYPE on page 13-91

parallel_clause Use the *parallel_clause* to parallelize creation of the domain index. For a nonpartitioned domain index, Oracle passes the explicit or default degree of parallelism to the ODCIIndexCreate cartridge routine, which in turn establishes parallelism for the index.

See Also: *Oracle9i Data Cartridge Developer's Guide* for complete information on the ODCI routines

PARAMETERS In the PARAMETERS clause, specify the parameter string that is passed uninterpreted to the appropriate ODCI indextype routine. The maximum length of the parameter string is 1000 characters.

When you specify this clause at the top level of the syntax, the parameters become the default parameters for the index partitions. If you specify this clause as part of the LOCAL [PARTITION] clause, you override any default parameters with parameters for the individual partition.

Once the domain index is created, Oracle invokes the appropriate ODCI routine. If the routine does not return successfully, the domain index is marked FAILED. The only operations supported on an failed domain index are DROP INDEX and (for non-local indexes) REBUILD INDEX.

See Also: *Oracle9i Data Cartridge Developer's Guide* for information on these routines

Restrictions on domain indexes:

- The *index_expr* can specify only a single column.
- You cannot specify a bitmap or unique domain index.

bitmap_join_index_clause

Use the *bitmap_join_index_clause* to define a **bitmap join index**. A bitmap join index is defined on a single table. For an index key made up of dimension table columns, it stores the fact table rowids corresponding to that key. In a data warehousing environment, the table on which the index is defined is commonly referred to as a **fact table**, and the tables with which this table is joined are commonly referred to as **dimension tables**. However, a star schema is not a requirement for creating a join index.

ON In the ON clause, first specify the fact table, and then inside the parentheses specify the columns of the dimension tables on which the index is defined.

FROM In the FROM clause, specify the joined tables.

WHERE In the WHERE clause, specify the join condition.

If the underlying fact table is partitioned, you must also specify one of the *local_partitioned_index* clauses (see *local_partitioned_index* on page 13-79).

Restrictions on bitmap join indexes: In addition to the restrictions on bitmap indexes in general (see BITMAP on page 13-69), the following restrictions apply to bitmap join indexes:

- You cannot create a bitmap join index on an index-organized table or a temporary table.
- No table may appear twice in the FROM clause.
- You cannot create a function-based join index.
- The dimension table columns must be either primary key columns or have unique constraints.
- If a dimension table has a composite primary key, each column in the primary key must be part of the join.
- You cannot specify the *local_index_clauses* unless the fact table is partitioned.

> **See Also:** *Oracle9i Data Warehousing Guide* for information on fact and dimension tables and on using bitmap indexes in a data warehousing environment

Examples

General Index Examples

Creating an Index: Example The following statement shows how the sample index ord_customer_ix on the customer_id column of the sample table oe.orders was created:

```
CREATE INDEX ord_customer_ix
   ON orders (customer_id);
```

Compressing an Index: Example To create the `ord_customer_ix` index with the `COMPRESS` clause, you might issue the following statement:

```
CREATE INDEX ord_customer_ix_demo
   ON orders (customer_id, sales_rep_id)
   COMPRESS 1;
```

The index will compress repeated occurrences of `customer_id` column values.

Computing Index Statistics: Example The following statement collects statistics on the `ord_customer_ix_demo` index during its creation:

```
CREATE INDEX ord_customer_ix_demo
   ON orders(customer_id, sales_rep_id)
   COMPUTE STATISTICS;
```

The type of statistics collected depends on the type of index you are creating.

Creating an Index in NOLOGGING Mode: Example If the sample table `orders` had been created using a fast parallel load (so all rows were already sorted), you could issue the following statement to quickly create an index.

```
/* Unless you first sort the table oe.orders, this example fails
   because you cannot specify NOSORT unless the base table is
   already sorted.
*/
CREATE INDEX ord_customer_ix_demo
   ON orders (order_mode)
   NOSORT
   NOLOGGING;
```

Creating a Cluster Index: Example To create an index for the `personnel` cluster, which was created in "Creating a Cluster: Example" on page 13-9, issue the following statement:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

No index columns are specified, because the index is automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

Create an Index on an XMLType Table: Example The following example creates an index on the area element of the `xwarehouses` table (created in "XMLType Table Examples" on page 15-71:

```
CREATE INDEX area_index ON xwarehouses e
```

```
   (EXTRACTVALUE(VALUE(e),'/Warehouse/Area'));
```

Such an index would greatly improve the performance of queries that select from the table based on, for example, the square footage of a warehouse, as shown in this statement:

```
SELECT e.getClobVal() AS warehouse
   FROM xwarehouses e
   WHERE EXISTSNODE(VALUE(e),'/Warehouse[Area>50000]') = 1;
```

See Also: EXISTSNODE on page 6-59 and VALUE on page 6-199

Function-Based Index Examples

Creating a Function-Based Index: Example The following statement creates a function-based index on the `employees` table based on an uppercase evaluation of the `last_name` column:

```
CREATE INDEX upper_ix ON employees (UPPER(last_name));
```

See the "Prerequisites" on page 13-62 for the privileges and parameter settings required when creating function-based indexes.

To ensure that Oracle will use the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, this statement is guaranteed to use the index:

```
SELECT first_name, last_name
   FROM employees WHERE UPPER(last_name) IS NOT NULL
   ORDER BY UPPER(last_name);
```

However, without the `WHERE` clause, Oracle may perform a full table scan.

In the next statements showing index creation and subsequent query, Oracle will use index `income_ix` even though the columns are in reverse order in the query:

```
CREATE INDEX income_ix
   ON employees(salary + (salary*commission_pct));

SELECT first_name||' '||last_name "Name"
   FROM employees
   WHERE (salary*commission_pct) + salary > 15000;
```

Creating a Function-Based Index on a LOB Column: Example The following statement uses the function created in "Using a Packaged Procedure in a Function: Example" on page 13-61 to create a function-based index on a LOB column in the

sample pm schema. The example then collects statistics on the function-based index and selects rows from the sample table print_media where that CLOB column has fewer than 1000 characters.

```
CREATE INDEX src_idx ON print_media(text_length(ad_sourcetext));

ANALYZE INDEX src_idx COMPUTE STATISTICS;

SELECT product_id FROM print_media
   WHERE text_length(ad_sourcetext) < 1000;

PRODUCT_ID
----------
      3060
      2056
      3106
      2268
```

Creating a Function-based Index on a Type Method: Example

This example entails an object type rectangle containing two number attributes: length and width. The area() method computes the area of the rectangle.

```
CREATE TYPE rectangle AS OBJECT
( length NUMBER,
  width NUMBER,
  MEMBER FUNCTION area RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY rectangle AS
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
   RETURN (length*width);
  END;
END;
```

Now, if you create a table rect_tab of type rectangle, you can create a function-based index on the area() method as follows:

```
CREATE TABLE rect_tab OF rectangle;
CREATE INDEX area_idx ON rect_tab x (x.area());
```

You can use this index efficiently to evaluate a query of the form:

```
SELECT * FROM rect_tab x WHERE x.area() > 100;
```

Partitioned Index Examples

Creating a Global Partitioned Index: Example The following statement creates a global prefixed index `amount_sold` on the sample table `sh.sales` with three partitions that divide the range of costs into three groups:

```
CREATE INDEX cost_ix ON sales (amount_sold)
   GLOBAL PARTITION BY RANGE (amount_sold)
      (PARTITION p1 VALUES LESS THAN (1000),
       PARTITION p2 VALUES LESS THAN (2500),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Creating an Index on a Hash-Partitioned Table: Example. The following statement creates a local index on the `product_id` column of the `product_information_part` partitioned table (which was created in"Hash Partitioning Example" on page 15-75) . The `STORE IN` clause immediately following `LOCAL` indicates that `product_information_part` is hash partitioned. Oracle will distribute the hash partitions between the `tbs1` and `tbs2` tablespaces:

```
CREATE INDEX prod_idx ON product_information_part(product_id) LOCAL
   STORE IN (tbs_1, tbs_2);
```

> **Note:** The creator of the index needs quote on the tablespaces specified. See CREATE TABLESPACE on page 15-80 for the examples that created these tablespaces.

Creating an Index on a Composite-Partitioned Table: Example. The following statement creates a local index on the `composite_sales` table, which was created in "Composite-Partitioned Table Examples" on page 15-75. The `STORAGE` clause specifies default storage attributes for the index. However, this default is overridden for the five subpartitions of partitions `q3_2000` and `q4_2000`, because separate `TABLESPACE` storage is specified.

> **Note:** The creator of the index must have quota on the tablespaces specified. See CREATE TABLESPACE on page 15-80 for the creation of tablespaces `tbs_1` and `tbs_2`.

```
CREATE INDEX sales_ix ON composite_sales(time_id, prod_id)
   STORAGE (INITIAL 1M MAXEXTENTS UNLIMITED)
   LOCAL
   (PARTITION q1_1998,
```

```
    PARTITION q2_1998,
    PARTITION q3_1998,
    PARTITION q4_1998,
    PARTITION q1_1999,
    PARTITION q2_1999,
    PARTITION q3_1999,
    PARTITION q4_1999,
    PARTITION q1_2000,
    PARTITION q2_2000
      (SUBPARTITION pq2001, SUBPARTITION pq2002,
       SUBPARTITION pq2003, SUBPARTITION pq2004,
       SUBPARTITION pq2005, SUBPARTITION pq2006,
       SUBPARTITION pq2007, SUBPARTITION pq2008),
    PARTITION q3_2000
      (SUBPARTITION c1 TABLESPACE tbs_1,
       SUBPARTITION c2 TABLESPACE tbs_1,
       SUBPARTITION c3 TABLESPACE tbs_1,
       SUBPARTITION c4 TABLESPACE tbs_1,
       SUBPARTITION c5 TABLESPACE tbs_1),
    PARTITION q4_2000
      (SUBPARTITION pq4001 TABLESPACE tbs_2,
       SUBPARTITION pq4002 TABLESPACE tbs_2,
       SUBPARTITION pq4003 TABLESPACE tbs_2,
       SUBPARTITION pq4004 TABLESPACE tbs_2)
);
```

Bitmap Index Example

The following creates a bitmap join index on the table `oe.product_information_part`, which was created in "Hash Partitioning Example" on page 15-75:

```
CREATE BITMAP INDEX product_bm_ix
   ON product_information_part(list_price)
   TABLESPACE tbs_1
   LOCAL(PARTITION ix_p1 TABLESPACE tbs_2,
         PARTITION ix_p2,
         PARTITION ix_p3 TABLESPACE tbs_3,
         PARTITION ix_p4,
         PARTITION ix_p5 TABLESPACE tbs_4 );
```

Because `product_information_part` is a partitioned table, the bitmap join index must be locally partitioned.

Note: In this example, the user must have quota on tablespaces specified. See CREATE TABLESPACE on page 15-80 for examples that create tablespaces tbs_2, tbs_3, and tbs_4.

Indexes on Nested Tables: Example

The sample table pm.print_media contains a nested table column ad_textdocs_ntab, which is stored in storage table textdocs_nestedtab. The following example creates a unique index on storage table textdocs_nestedtab:

```
CREATE UNIQUE INDEX nested_tab_ix
      ON textdocs_nestedtab(NESTED_TABLE_ID, document_typ);
```

Including pseudocolumn NESTED_TABLE_ID ensures distinct rows in nested table column ad_textdocs_ntab.

Indexing on Substitutable Columns: Examples

You can build an index on attributes of the declared type of a substitutable column. In addition, you can reference the subtype attributes by using the appropriate TREAT function. The following example uses the table books, which is created in "Substitutable Table and Column Examples" on page 15-67. The statement creates an index on the salary attribute of all employee authors in the books table:

```
CREATE INDEX salary_i
   ON books (TREAT(author AS employee_t).salary);
```

The target type in the argument of the TREAT function must be the type that added the attribute being referenced. In the example, the target of TREAT is employee_t, which is the type that added the salary attribute.

If this condition is not satisfied, Oracle interprets the TREAT function as any functional expression and creates the index as a function-based index. For example, the following statement creates a function-based index on the salary attribute of part-time employees, assigning nulls to instances of all other types in the type hierarchy.

```
CREATE INDEX salary_func_i ON persons p
   (TREAT(VALUE(P) AS part_time_emp_t).salary);
```

You can also build an index on the type-discriminant column underlying a substitutable column by using the SYS_TYPEID function.

Note: Oracle uses the type-discriminant column to evaluate queries that involve the `IS OF` *`type`* condition. The cardinality of the typeid column is normally low, so Oracle Corporation recommends that you build a bitmap index in this situation.

The following statement creates a bitmap index on the typeid of the author column of the books table:

```
CREATE BITMAP INDEX typeid_i ON books (SYS_TYPEID(author));
```

See Also:

- "Type Hierarchy Example" on page 16-22 to see the creation of the type hierarchy underlying the `books` table
- TREAT on page 6-188
- SYS_TYPEID on page 6-161
- "IS OF type Conditions" on page 5-19

CREATE INDEXTYPE

Purpose

Use the CREATE INDEXTYPE statement to create an **indextype**, which is an object that specifies the routines that manage a domain (application-specific) index. Indextypes reside in the same namespace as tables, views, and other schema objects. This statement binds the indextype name to an implementation type, which in turn specifies and refers to user-defined index functions and procedures that implement the indextype.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* and *Oracle9i Database Concepts* for more information on implementing indextypes

Prerequisites

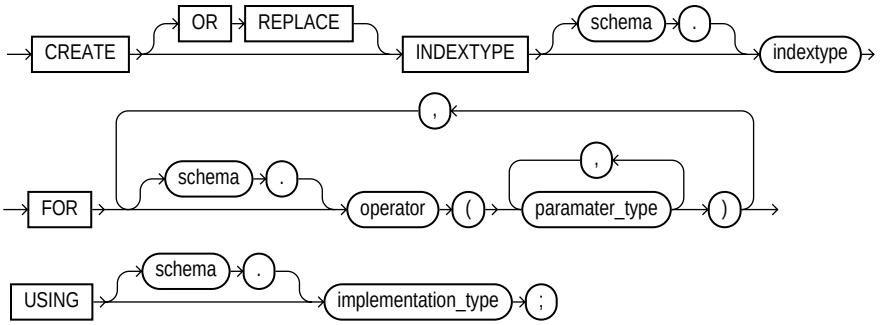
To create an indextype in your own schema, you must have the CREATE INDEXTYPE system privilege. To create an indextype in another schema, you must have CREATE ANY INDEXTYPE system privilege. In either case, you must have the EXECUTE object privilege on the implementation type and the supported operators.

An indextype supports one or more operators, so before creating an indextype, you should first design the operator or operators to be supported and provide functional implementation for those operators.

> **See Also:** CREATE OPERATOR on page 14-42

Syntax

create_indextype::=



Keywords and Parameters

schema

Specify the name of the schema in which the indextype resides. If you omit *schema*, Oracle creates the indextype in your own schema.

indextype

Specify the name of the indextype to be created.

FOR Clause

Use the FOR clause to specify the list of operators supported by the indextype.

- For *schema*, specify the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.
- For *operator*, specify the name of the operator supported by the indextype.

  All the operators listed in this clause should be valid operators.
- For *parameter_type*, list the types of parameters to the operator.

USING Clause

The USING clause lets you specify the type that provides the implementation for the new indextype.

For *implementation_type*, specify the name of the type that implements the appropriate Oracle Data Cartridge interface (ODCI).

- You must specify a valid type that implements the routines in the ODCI interface.
- The implementation type must reside in the same schema as the indextype.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for additional information on this interface

Example

Creating an Indextype: Example The following statement creates an indextype named `TextIndexType` and specifies the `contains` operator that is supported by the indextype and the `TextIndexMethods` type that implements the index interface:

```
CREATE INDEXTYPE TextIndexType
   FOR contains (VARCHAR2, VARCHAR2)
   USING TextIndexMethods;
```

CREATE JAVA

Purpose

Use the CREATE JAVA statement to create a schema object containing a Java source, class, or resource.

See Also:

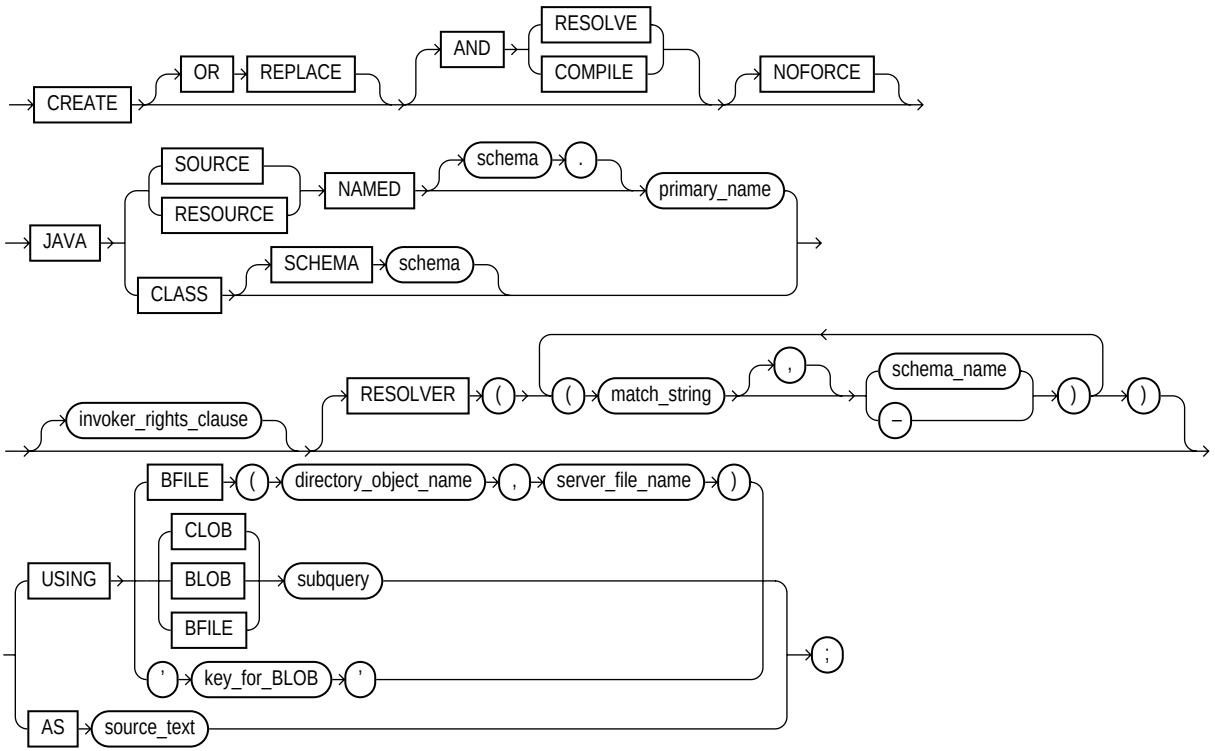
- *Oracle9i Java Developer's Guide* for Java concepts
- *Oracle9i Java Stored Procedures Developer's Guide* for Java stored procedures
- *Oracle9i SQLJ Developer's Guide and Reference* for SQLJ
- *Oracle9i JDBC Developer's Guide and Reference* for JDBC

Prerequisites

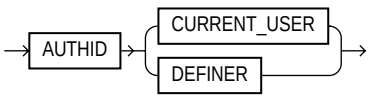
To create or replace a schema object containing a Java source, class, or resource in your own schema, you must have CREATE PROCEDURE system privilege. To create such a schema object in another user's schema, you must have CREATE ANY PROCEDURE system privilege. To replace such a schema object in another user's schema, you must also have ALTER ANY PROCEDURE system privilege.

Syntax

create_java::=



invoker_rights_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the schema object containing the Java class, source, or resource if it already exists. Use this clause to change the definition of an existing object without dropping, re-creating, and regranting object privileges previously granted.

If you redefine a Java schema object and specify RESOLVE or COMPILE, Oracle recompiles or resolves the object. Whether or not the resolution or compilation is successful, Oracle invalidates classes that reference the Java schema object.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

See Also: ALTER JAVA on page 9-89 for additional information

RESOLVE | COMPILE

RESOLVE and COMPILE are synonymous keywords. They specify that Oracle should attempt to resolve the Java schema object that is created if this statement succeeds.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

Restriction on RESOLVE and COMPILE: You cannot specify this clause for a Java resource.

NOFORCE

Specify NOFORCE to roll back the results of this CREATE command if you have specified either RESOLVE or COMPILE, and the resolution or compilation fails. If you do not specify this option, Oracle takes no action if the resolution or compilation fails (that is, the created schema object remains).

JAVA SOURCE Clause

Specify JAVA SOURCE to load a Java source file.

JAVA CLASS Clause

Specify JAVA CLASS to load a Java class file.

JAVA RESOURCE Clause

Specify JAVA RESOURCE to load a Java resource file.

NAMED Clause

The NAMED clause is *required* for a Java source or resource. The *primary_name* must be enclosed in double quotation marks.

- For a Java source, this clause specifies the name of the schema object in which the source code is held. A successful CREATE JAVA SOURCE statement will also create additional schema objects to hold each of the Java classes defined by the source.
- For a Java resource, this clause specifies the name of the schema object to hold the Java resource.

Use double quotation marks to preserve a lower- or mixed-case *primary_name*.

If you do not specify *schema*, Oracle creates the object in your own schema.

Restrictions on the NAMED clause:

- You cannot specify NAMED for a Java class.
- The *primary_name* cannot contain a database link.

SCHEMA Clause

The SCHEMA clause applies only to a Java class. This optional clause specifies the schema in which the object containing the Java file will reside. If you do not specify this clause, Oracle creates the object in your own schema.

invoker_rights_clause

Use the *invoker_rights_clause* to indicate whether the methods of the class execute with the privileges and in the schema of the user who owns the class or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

AUTHID CURRENT_USER

CURRENT_USER indicates that the methods of the class execute with the privileges of CURRENT_USER. This clause is the default and creates an **invoker-rights class**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER

DEFINER indicates that the methods of the class execute with the privileges of the owner of the schema in which the class resides, and that external names resolve in the schema where the class resides. This clause creates a **definer-rights class**.

See Also:

- *Oracle9i Java Stored Procedures Developer's Guide*
- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined

RESOLVER Clause

The RESOLVER clause lets you specify a mapping of the fully qualified Java name to a Java schema object, where

- *match_string* is either a fully qualified Java name, a wildcard that can match such a Java name, or a wildcard that can match any name.
- *schema_name* designates a schema to be searched for the corresponding Java schema object.
- A dash (-) as an alternative to *schema_name* indicates that if *match_string* matches a valid Java name, Oracle can leave the name unresolved. The resolution succeeds, but the name cannot be used at run time by the class.

This mapping is stored with the definition of the schema objects created in this command for use in later resolutions (either implicit or in explicit ALTER ... RESOLVE statements).

USING Clause

The USING clause determines a sequence of character (CLOB or BFILE) or binary (BLOB or BFILE) data for the Java class or resource. Oracle uses the sequence of characters to define one file for a Java class or resource, or one source file and one or more derived classes for a Java source.

BFILE Clause

Specify the directory and filename of a previously created file on the operating system (*directory_object_name*) and server file (*server_file_name*) containing the sequence. BFILE is usually interpreted as a character sequence by CREATE JAVA SOURCE and as a binary sequence by CREATE JAVA CLASS or CREATE JAVA RESOURCE.

CLOB | BLOB | BFILE *subquery*

Specify a query that selects a single row and column of the type specified (CLOB, BLOB, or BFILE). The value of the column makes up the sequence of characters.

Note: In earlier releases, the USING clause implicitly supplied the keyword SELECT. This is no longer the case. However, the subquery without the keyword SELECT is still supported for backward compatibility.

key_for_BLOB

The *key_for_BLOB* clause supplies the following implicit query:

```
SELECT LOB FROM CREATE$JAVA$LOB$TABLE
   WHERE NAME = 'key_for_BLOB';
```

Restriction on the *key_for_BLOB* clause: To use this case, the table CREATE$JAVA$LOB$TABLE must exist in the current schema and must have a column LOB of type BLOB and a column NAME of type VARCHAR2.

AS *source_text*

Specify a sequence of characters for a Java or SQLJ source.

Examples

Creating a Java Class Object: Example The following statement creates a schema object containing a Java class using the name found in a Java binary file:

```
CREATE JAVA CLASS USING BFILE (bfile_dir, 'Agent.class');
```

This example assumes the directory object bfile_dir, which points to the operating system directory containing the Java class Agent.class, already exists. In this example, the name of the class determines the name of the Java class schema object.

Creating a Java Source Object: Example The following statement creates a Java source schema object:

```
CREATE JAVA SOURCE NAMED "Hello" AS
   public class Hello {
      public static String hello() {
         return "Hello World";   } };
```

Creating a Java Resource Object: Example The following statement creates a Java resource schema object named apptext from a bfile:

```
CREATE JAVA RESOURCE NAMED "appText"
   USING BFILE (bfile_dir, 'textBundle.dat');
```

14

SQL Statements: CREATE LIBRARY to CREATE SPFILE

This chapter contains the following SQL statements:

- CREATE LIBRARY
- CREATE MATERIALIZED VIEW
- CREATE MATERIALIZED VIEW LOG
- CREATE OPERATOR
- CREATE OUTLINE
- CREATE PACKAGE
- CREATE PACKAGE BODY
- CREATE PFILE
- CREATE PROCEDURE
- CREATE PROFILE
- CREATE ROLE
- CREATE ROLLBACK SEGMENT
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE SPFILE

CREATE LIBRARY

Purpose

Use the CREATE LIBRARY statement to create a schema object associated with an operating-system shared library. The name of this schema object can then be used in the *call_spec* of CREATE FUNCTION or CREATE PROCEDURE statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can call to third-generation-language (3GL) functions and procedures.

> **See Also:** CREATE FUNCTION on page 13-49 and *PL/SQL User's Guide and Reference* for more information on functions and procedures

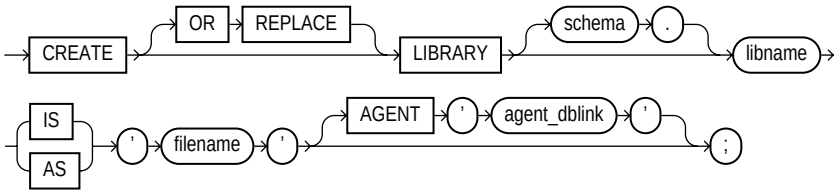
Prerequisites

To create a library in your own schema, you must have the CREATE LIBRARY system privilege. To create a library in another user's schema, you must have the CREATE ANY LIBRARY system privilege. To use the procedures and functions stored in the library, you must have EXECUTE object privileges on the library.

The CREATE LIBRARY statement is valid only on platforms that support shared libraries and dynamic linking.

Syntax

create_library::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the library if it already exists. Use this clause to change the definition of an existing library without dropping, re-creating, and regranting schema object privileges granted on it.

Users who had previously been granted privileges on a redefined library can still access the library without being regranted the privileges.

libname

Specify the name you wish to represent this library when declaring a function or procedure with a *call_spec*.

'*filename*'

Specify a string literal, enclosed in single quotes. This string should be the path or filename your operating system recognizes as naming the shared library.

The '*filename*' is not interpreted during execution of the CREATE LIBRARY statement. The existence of the library file is not checked until an attempt is made to execute a routine from it.

AGENT Clause

Specify the AGENT clause if you want external procedures to be run from a database link other than the server. Oracle will use the database link specified by *agent_dblink* to run external procedures. If you omit this clause, the default agent on the server (extproc) will run external procedures.

Examples

Creating a Library: Examples The following statement creates library ext_lib:

```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
/
```

The following statement re-creates library ext_lib:

```
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
/
```

Specifying an External Procedure Agent: Example The following example creates a library app_lib and specifies that external procedures will be run from the public database sales.hq.acme.com:

```
CREATE LIBRARY app_lib as '${ORACLE_HOME}/lib/app_lib.so'
   AGENT 'sales.hq.acme.com';
/
```

See Also: "Defining a Public Database Link: Example" on page 13-39 for information on creating database links

CREATE MATERIALIZED VIEW

Purpose

Use the CREATE MATERIALIZED VIEW statement to create a **materialized view**. A materialized view is a database object that contains the results of a query. The FROM clause of the query can name tables, views, and other materialized views. Collectively these are called **master tables** (a replication term) or **detail tables** (a data warehouse term). This reference uses "master tables" for consistency. The databases containing the master tables are called the **master databases**.

Note: The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

For replication purposes, materialized views allow you to maintain copies of remote data on your local node. The copies can be updatable with the Advanced Replication feature and are read-only without this feature. You can select data from a materialized view as you would from a table or view. In replication environments, the materialized views commonly created are **primary key**, **rowid**, **object**, and **subquery** materialized views.

> **See Also:** *Oracle9i Replication* for information on the types of materialized views used to support replication

For data warehousing purposes, the materialized views commonly created are **materialized aggregate views**, **single-table materialized aggregate views**, and **materialized join views**. All three types of materialized views can be used by query rewrite, an optimization technique that transforms a user request written in terms of master tables into a semantically equivalent request that includes one or more materialized views.

> **See Also:**
>
> - ALTER MATERIALIZED VIEW on page 9-92
> - *Oracle9i Data Warehousing Guide* for information on the types of materialized views used to support data warehousing

Prerequisites

The privileges required to create a materialized view should be granted directly rather than through a role.

To create a materialized view **in your own schema**:

- You must have been granted the CREATE MATERIALIZED VIEW system privilege **an**d either the CREATE TABLE or CREATE ANY TABLE system privilege.
- You must also have access to any master tables of the materialized view that you do not own, either through a SELECT object privilege on each of the tables or through the SELECT ANY TABLE system privilege.

To create a materialized view **in another user's schema**:

- You must have the CREATE ANY MATERIALIZED VIEW system privilege.
- The owner of the materialized view must have the CREATE TABLE system privilege. The owner must also have access to any master tables of the materialized view that the schema owner does not own (for example, if the master tables are on a remote database), *and* to any materialized view logs defined on those master tables, either through a SELECT object privilege on each of the tables or through the SELECT ANY TABLE system privilege.

To create a refresh-on-commit materialized view (ON COMMIT REFRESH clause), in addition to the preceding privileges, you must have the ON COMMIT REFRESH object privilege on any master tables that you do not own or you must have the ON COMMIT REFRESH system privilege.

To create the materialized view **with query rewrite enabled**, in addition to the preceding privileges:

- The owner of the master tables must have the QUERY REWRITE system privilege.
- If you are not the owner of the master tables, you must have the GLOBAL QUERY REWRITE system privilege or the QUERY REWRITE object privilege on each table outside your schema.
- If the schema owner does not own the master tables, then the schema owner must have the GLOBAL QUERY REWRITE privilege or the QUERY REWRITE object privilege on each table outside the schema.
- If you are defining the materialized view on a prebuilt container (ON PREBUILT TABLE), you must have the SELECT privilege WITH GRANT OPTION on the container table.

The user whose schema contains the materialized view must have sufficient quota in the target tablespace to store the materialized view's master table and index or must have the `UNLIMITED TABLESPACE` system privilege.

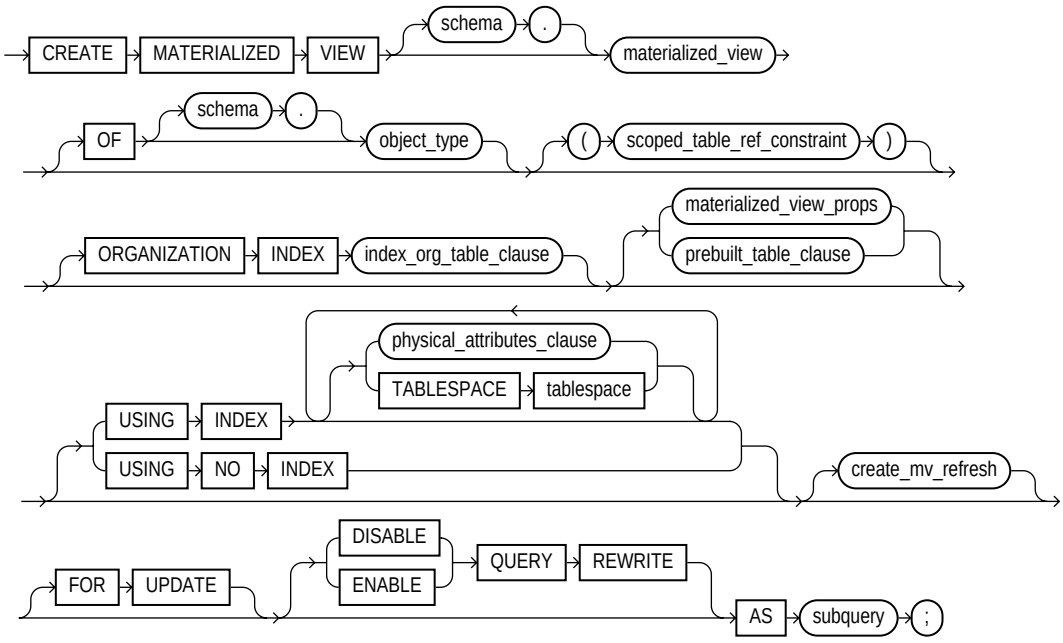
When you create a materialized view, Oracle creates one internal table and at least one index, and may create one view, all in the schema of the materialized view. Oracle uses these objects to maintain the materialized view's data. You must have the privileges necessary to create these objects.

See Also:

- CREATE TABLE on page 15-7, CREATE VIEW on page 16-39, and CREATE INDEX on page 13-62 for information on these privileges
- *Oracle9i Replication* for information about the prerequisites that apply to creating replication materialized views
- *Oracle9i Data Warehousing Guide* for information about the prerequisites that apply to creating data warehousing materialized views

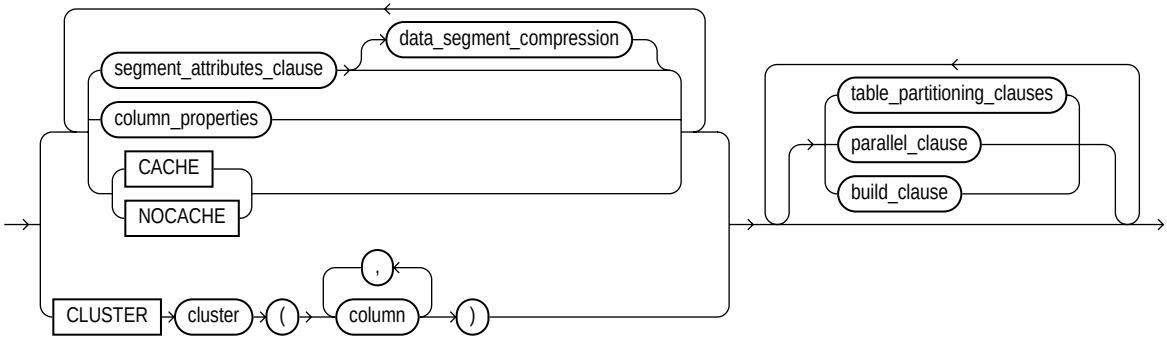
Syntax

create_materialized_view::=



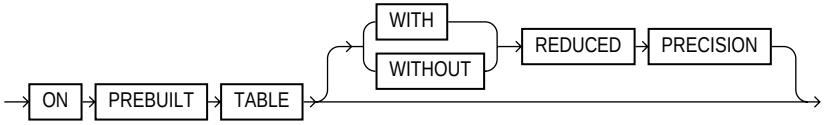
(*scoped_table_ref_constraint::=* on page 14-8, *index_org_table_clause::=* on page 14-9, *materialized_view_props::=* on page 14-8, *prebuilt_table_clause::=* on page 14-8, *physical_attributes_clause::=* on page 14-11, *create_mv_refresh::=* on page 14-10, *subquery::=* on page 18-5)

materialized_view_props::=

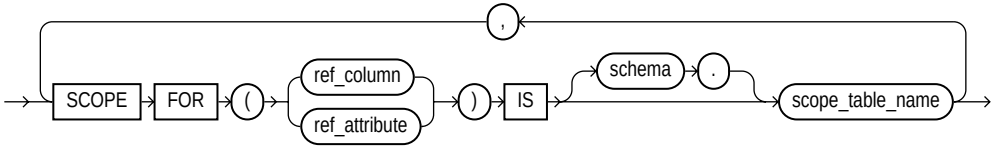


(*segment_attributes_clause::=* on page 14-10, *data_segment_compression::=* on page 14-11, *column_properties::=* on page 14-11, *table_partitioning_clauses* on page 15-44—part of CREATE TABLE syntax, *parallel_clause::=* on page 14-14, *build_clause::=* on page 14-14)

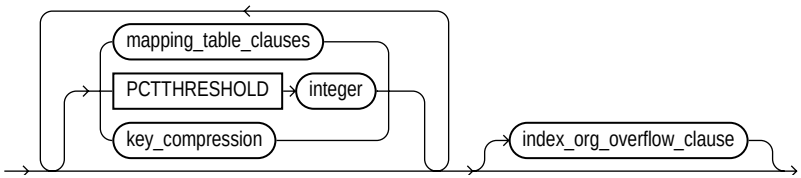
prebuilt_table_clause::=



scoped_table_ref_constraint::=

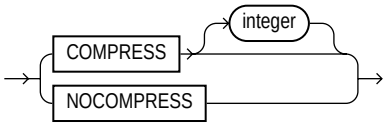


index_org_table_clause::=

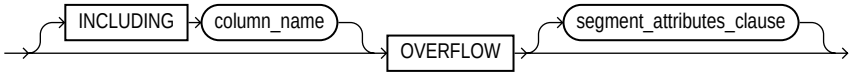


(*mapping_table_clauses*: not supported with materialized views, *key_compression::=* on page 14-9, *index_org_overflow_clause::=* on page 14-9)

key_compression::=

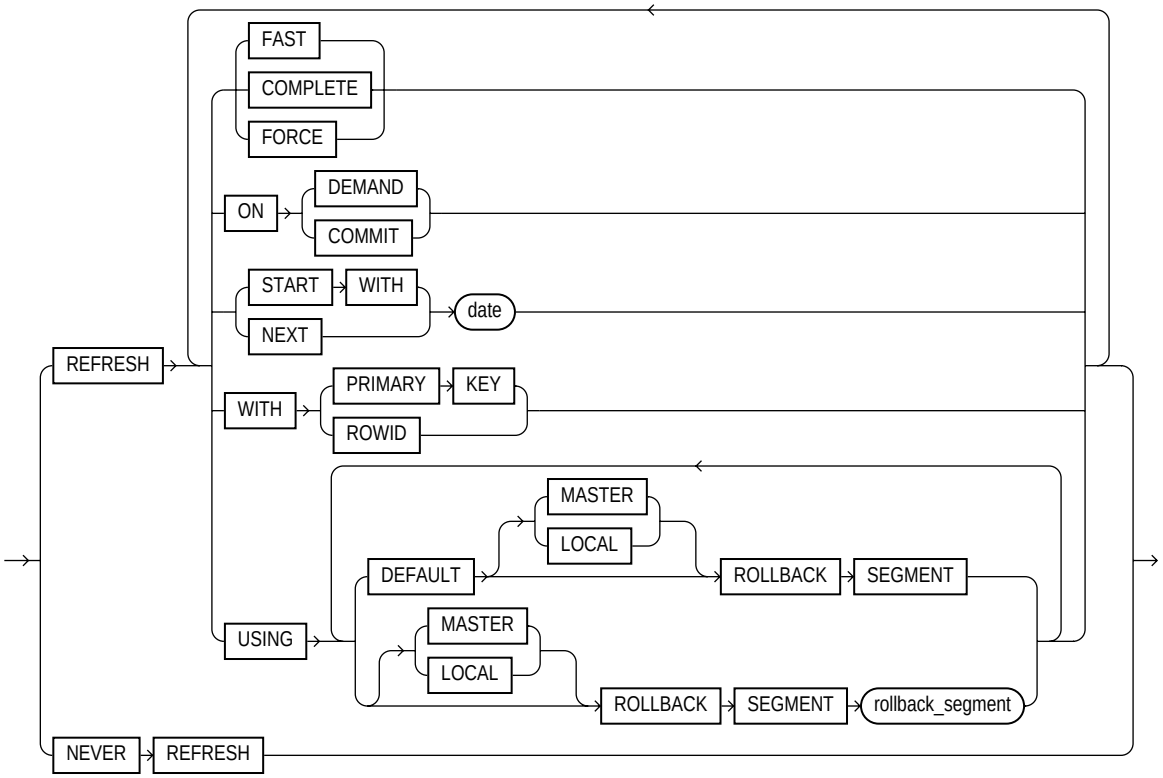


index_org_overflow_clause::=

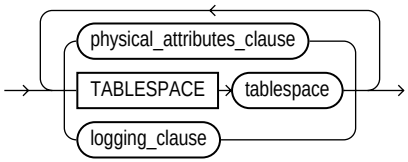


(*segment_attributes_clause::=* on page 14-10)

create_mv_refresh::=

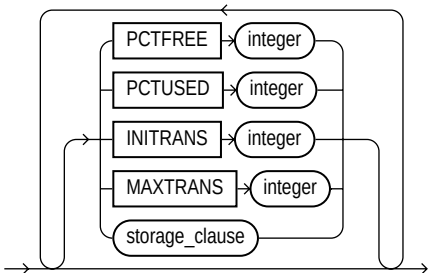


segment_attributes_clause::=



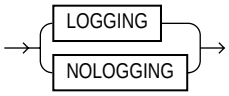
(*physical_attributes_clause::=* on page 14-11, *logging_clause::=* on page 14-11)

physical_attributes_clause::=

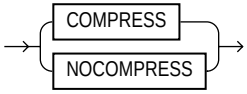


(*logging_clause::=* on page 7-46)

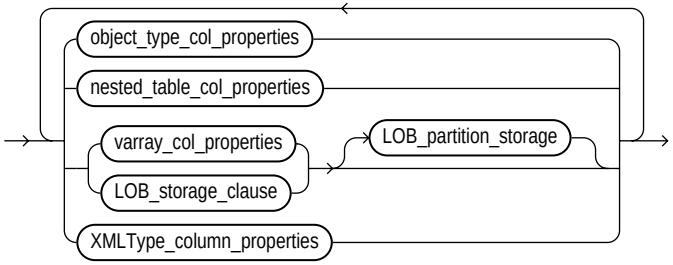
logging_clause::=



data_segment_compression::=

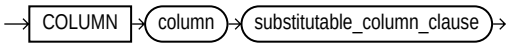


column_properties::=



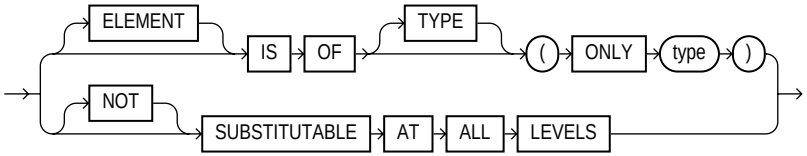
(*object_type_col_properties::=* on page 14-12, *nested_table_col_properties::=* on page 14-12, *varray_col_properties::=* on page 14-12, *LOB_partition_storage::=* on page 14-14, *LOB_storage_clause::=* on page 14-13, XMLType_column_properties: not supported for materialized views)

object_type_col_properties::=

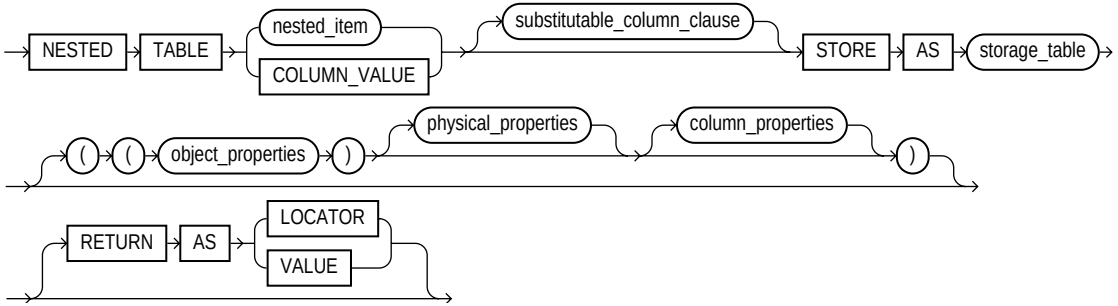


(*substitutable_column_clause::=* on page 14-12)

substitutable_column_clause::=

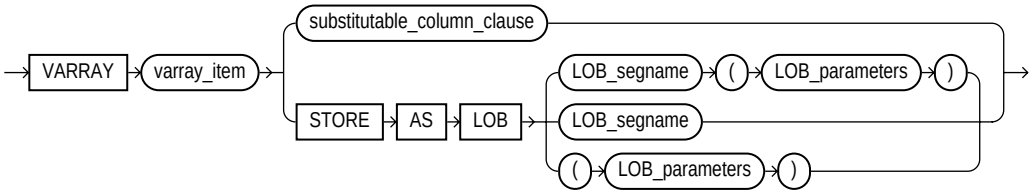


nested_table_col_properties::=



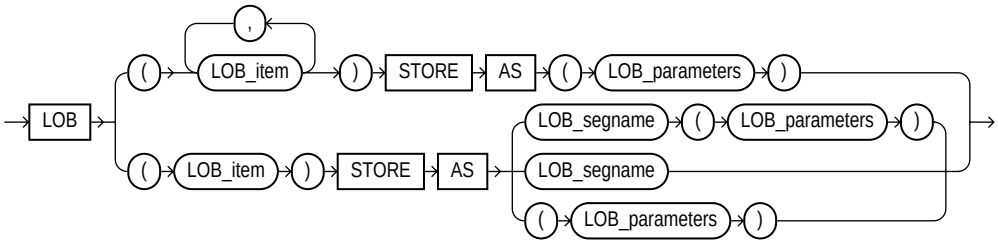
(*substitutable_column_clause::=* on page 14-12, *object_properties::=* on page 15-10, *physical_properties::=* on page 15-11—part of CREATE TABLE syntax, *column_properties::=* on page 14-11)

varray_col_properties::=



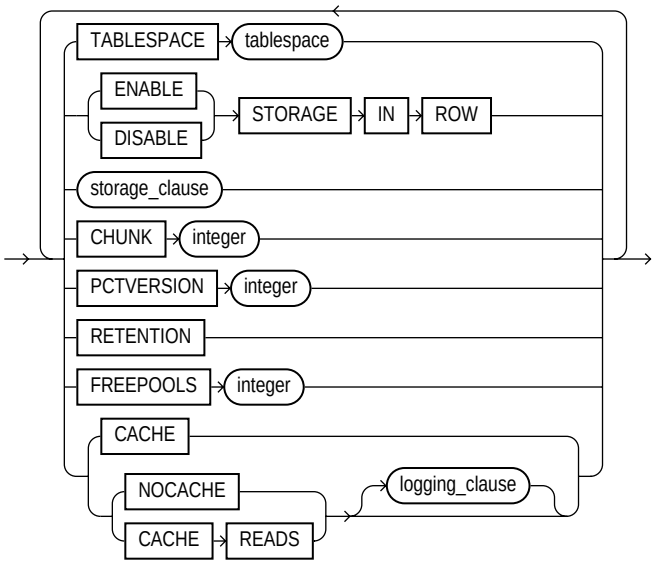
(*substitutable_column_clause::=* on page 14-12, *LOB_parameters::=* on page 14-13)

LOB_storage_clause::=



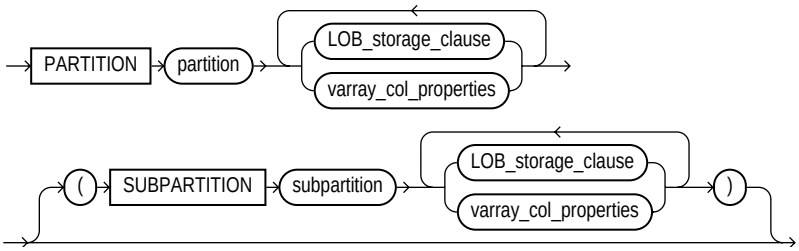
(*LOB_parameters::=* on page 14-13)

LOB_parameters::=



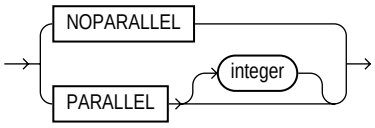
(*storage_clause::= on page 7-58*, *logging_clause::=* on page 14-11)

LOB_partition_storage::=

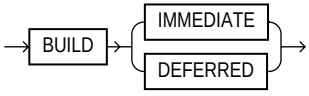


(*LOB_storage_clause::=* on page 14-13, *varray_col_properties::=* on page 14-12)

parallel_clause::=



build_clause::=



Keywords and Parameters

schema

Specify the schema to contain the materialized view. If you omit *schema*, Oracle creates the materialized view in your schema.

materialized_view

Specify the name of the materialized view to be created. Oracle generates names for the table and indexes used to maintain the materialized view by adding a prefix or suffix to the materialized view name.

OF *type_name*

The OF *type_name* clause lets you explicitly create an **object materialized view** of type *object_type*.

> **See Also:** See the *object_table* clause of CREATE TABLE on page 15-52 for more information on the OF *type_name* clause

scoped_table_ref_constraint

Use the SCOPE FOR clause to restrict the scope of references to a single table, *scope_table_name*. The values in the REF column or attribute point to objects in *scope_table_name*, in which object instances (of the same type as the REF column) are stored.

> **See Also:** "SCOPE REF Constraints" on page 7-17 for more information

index_org_table_clause

The ORGANIZATION INDEX clause lets you create an index-organized materialized view. In such a materialized view, data rows are stored in an index defined on the primary key of the materialized view. You can specify index organization for the following types of materialized views:

- Read-only and updatable object materialized views. You must ensure that the master table has a primary key.
- Read-only and updatable primary key based materialized views
- Read-only rowid materialized views.

The keywords and parameters of the *index_org_table_clause* have the same semantics as described in CREATE TABLE, with the restrictions that follow.

> **See Also:** the *index_org_table_clause* of CREATE TABLE on page 15-31

Restrictions on index-organized materialized views:

- You cannot specify the following CREATE MATERIALIZED VIEW clauses: CACHE or NOCACHE, CLUSTER, or ON PREBUILT TABLE.
- In the *index_org_table_clause*:
  - You cannot specify the *mapping_table_clauses*.
  - You can specify COMPRESS only for a materialized view based on a composite primary key. You can specify NOCOMPRESS for a materialized view based on either a simple or composite primary key.

materialized_view_props

Use these property clauses to describe a materialized view that is not based on an existing table. To create a materialized view that is based on an existing table, use the *prebuilt_table_clause* on page 14-19.

segment_attributes_clause

Use the *segment_attributes_clause* to establish values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters (or, when used in the USING INDEX clause, for the INITRANS and MAXTRANS parameters only), the storage characteristics for the materialized view, to assign a tablespace, and to specify whether logging is to occur.

See Also:

- *physical_attributes_clause* on page 7-52 for a complete description of the parameters of the *physical_attributes_clause*, including default values
- *storage_clause* on page 7-56 for a complete description of the *storage_clause*, including default values

data_segment_compression

Use the *data_segment_compression* clause to instruct Oracle whether to compress data segments to reduce disk and memory use. The COMPRESS keyword enables data segment compression. The NOCOMPRESS keyword disables data segment compression.

See Also: *data_segment_compression* clause of CREATE TABLE on page 15-29 for more information on data segment compression

TABLESPACE Clause

Specify the tablespace in which the materialized view is to be created. If you omit this clause, Oracle creates the materialized view in the default tablespace of the schema containing the materialized view.

column_properties

The *column_properties* clause lets you specify the storage characteristics of a LOB, nested table, varray, or XMLType column.

Restriction on *column_properties*: The *object_type_col_properties* are not relevant for a materialized view.

> **See Also:** CREATE TABLE on page 15-7 for detailed information about specifying the parameters of this clause

logging_clause

Specify LOGGING or NOLOGGING to establish the logging characteristics for the materialized view. The default is the logging characteristic of the tablespace in which the materialized view resides.

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this table are placed at the *most recently used* end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. NOCACHE specifies that the blocks are placed at the *least recently used* end of the LRU list.

Note: NOCACHE has no effect on materialized views for which you specify KEEP in the *storage_clause*.

> **See Also:** CREATE TABLE on page 15-7 for information about specifying CACHE or NOCACHE

CLUSTER Clause

Use the CLUSTER clause to create the materialized view as part of the specified cluster. A clustered materialized view uses the cluster's space allocation. Therefore, do not specify the *physical_attributes_clause* or the TABLESPACE clause with the CLUSTER clause.

table_partitioning_clauses

The *table_partitioning_clauses* let you specify that the materialized view is partitioned on specified ranges of values or on a hash function. Partitioning of materialized views is the same as partitioning of tables.

> **See Also:** *table_partitioning_clauses* of CREATE TABLE on page 15-44

parallel_clause

The *parallel_clause* lets you indicate whether parallel operations will be supported for the materialized view and sets the default degree of parallelism for queries and DML on the materialized view after creation.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

build_clause

The *build_clause* lets you specify when to populate the materialized view.

IMMEDIATE Specify IMMEDIATE to indicate that the materialized view is populated immediately. This is the default.

DEFERRED Specify DEFERRED to indicate that the materialized view will be populated by the next REFRESH operation. The first (deferred) refresh must always

be a complete refresh. Until then, the materialized view has a staleness value of UNUSABLE, so it cannot be used for query rewrite.

prebuilt_table_clause

The ON PREBUILT TABLE clause lets you register an existing table as a preinitialized materialized view. This is particularly useful for registering large materialized views in a data warehousing environment. The table must have the same name and be in the same schema as the resulting materialized view.

If the materialized view is dropped, the preexisting table reverts to its identity as a table.

> **Caution:** This clause assumes that the table object reflects the materialization of a subquery. Oracle Corporation strongly recommends that you ensure that this assumption is true in order to ensure that the materialized view correctly reflects the data in its master tables.

WITH REDUCED PRECISION Specify WITH REDUCED PRECISION to authorize the loss of precision that will result if the precision of the table or materialized view columns do not exactly match the precision returned by *subquery*.

WITHOUT REDUCED PRECISION Specify WITHOUT REDUCED PRECISION to require that the precision of the table or materialized view columns match exactly the precision returned by *subquery*, or the create operation will fail. This is the default.

Restrictions on *prebuilt_table_clause*:

- Each column alias in *subquery* must correspond to a column in *table_name*, and corresponding columns must have matching datatypes.
- If you specify this clause, you cannot specify a NOT NULL constraint for any column that is unmanaged (that is, not referenced in *subquery*) unless you also specify a default value for that column.

> **See Also:** "Creating Prebuilt Materialized Views: Example" on page 14-28

USING INDEX Clause

The USING INDEX clause lets you establish the value of INITRANS, MAXTRANS, and STORAGE parameters for the default index Oracle uses to maintain the materialized view's data. If USING INDEX is not specified, then default values are used for the index. Oracle uses the default index to speed up incremental ("fast") refresh of the materialized view.

Restriction on USING INDEX clause: You cannot specify the PCTUSED or PCTFREE parameters in this clause.

USING NO INDEX Clause

Specify USING NO INDEX to suppress the creation of the default index. You can create an alternative index explicitly by using the CREATE INDEX statement. You should create such an index if you specify USING NO INDEX and you are creating the materialized view with the incremental refresh method (REFRESH FAST).

create_mv_refresh

Use the *create_mv_refresh* to specify the default methods, modes, and times for Oracle to refresh the materialized view. If the master tables of a materialized view are modified, the data in the materialized view must be updated to make the materialized view accurately reflect the data currently in its master tables. This clause lets you schedule the times and specify the method and mode for Oracle to refresh the materialized view.

Note: This clause only sets the default refresh options. For instructions on actually implementing the refresh, refer to *Oracle9i Replication* and *Oracle9i Data Warehousing Guide*.

See Also: "Periodic Refresh of Materialized Views: Example" on page 14-29 and "Automatic Refresh Times for Materialized Views: Example" on page 14-29

FAST Clause

Specify FAST to indicate the incremental refresh method, which performs the refresh according to the changes that have occurred to the master tables. The changes are stored either in the materialized view log associated with the master table (for conventional DML changes) or in the direct loader log (for direct-path INSERT operations).

If you specify REFRESH FAST, the CREATE statement will fail unless materialized view logs already exist for the materialized view's master tables. (Oracle creates the direct loader log automatically when a direct-path INSERT takes place. No user intervention is needed.)

For both conventional DML changes and for direct-path INSERT operations, other conditions may restrict the eligibility of a materialized view for fast refresh.

Materialized views are not eligible for fast refresh if the defining subquery contains an analytic function.

See Also:

- *Oracle9i Replication* for restrictions on fast refresh in replication environments
- *Oracle9i Data Warehousing Guide* for restrictions on fast refresh in data warehouse environments
- "Analytic Functions" on page 6-9
- "Creating a Fast Refreshable Materialized View: Example" on page 14-31

COMPLETE Clause

Specify COMPLETE to indicate the complete refresh method, which is implemented by executing the materialized view's defining subquery. If you request a complete refresh, Oracle performs a complete refresh even if a fast refresh is possible.

FORCE Clause

Specify FORCE to indicate that when a refresh occurs, Oracle will perform a fast refresh if one is possible or a complete refresh otherwise. If you do not specify a refresh method (FAST, COMPLETE, or FORCE), FORCE is the default.

ON COMMIT Clause

Specify ON COMMIT to indicate that a fast refresh is to occur whenever Oracle commits a transaction that operates on a master table of the materialized view. This clause may increase the time taken to complete the commit, because Oracle performs the refresh operation as part of the commit process.

Restriction on ON COMMIT: This clause is not supported for materialized views containing object types.

See Also: *Oracle9i Replication* and *Oracle9i Data Warehousing Guide*

ON DEMAND Clause

Specify ON DEMAND to indicate that the materialized view will be refreshed on demand by calling one of the three DBMS_MVIEW refresh procedures. If you omit both ON COMMIT and ON DEMAND, ON DEMAND is the default.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on these procedures
- *Oracle9i Data Warehousing Guide* on the types of materialized views you can create by specifying REFRESH ON DEMAND

If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.

START WITH Clause

Specify a date expression for the first automatic refresh time.

NEXT Clause

Specify a date expression for calculating the interval between automatic refreshes.

Both the START WITH and NEXT values must evaluate to a time in the future. If you omit the START WITH value, Oracle determines the first automatic refresh time by evaluating the NEXT expression with respect to the creation time of the materialized view. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the materialized view only once. If you omit both the START WITH and NEXT values, or if you omit the *create_mv_refresh* entirely, Oracle does not automatically refresh the materialized view.

WITH PRIMARY KEY Clause

Specify WITH PRIMARY KEY to create a primary key materialized view. This is the default, and should be used in all cases except those described for WITH ROWID. Primary key materialized views allow materialized view master tables to be reorganized without affecting the eligibility of the materialized view for fast refresh. The master table must contain an enabled primary key constraint.

Restriction on WITH PRIMARY KEY: You cannot specify this clause for an object materialized view. Oracle implicitly refreshes object materialized WITH OBJECT ID.

> **See Also:** *Oracle9i Replication* for detailed information about primary key materialized views and "Creating Primary Key Materialized Views: Example" on page 14-29

WITH ROWID Clause

Specify `WITH ROWID` to create a rowid materialized view. Rowid materialized views provide compatibility with master tables in releases of Oracle prior to 8.0.

You can also use rowid materialized views if the materialized view does not include all primary key columns of the master tables. Rowid materialized views must be based on a single table and cannot contain any of the following:

- Distinct or aggregate functions
- `GROUP BY` or `CONNECT BY` clauses
- Subqueries
- Joins
- Set operations

Rowid materialized views are not eligible for fast refreshed after a master table reorganization until a complete refresh has been performed.

Restriction on WITH ROWID: You cannot specify this clause for an object materialized view. Oracle implicitly refreshes object materialized `WITH OBJECT ID`.

> **See Also:** "Creating Materialized Aggregate Views: Example" on page 14-27 and "Creating Rowid Materialized Views: Example" on page 14-29

USING ROLLBACK SEGMENT Clause

Specify the remote rollback segment to be used during materialized view refresh, where *`rollback_segment`* is the name of the rollback segment to be used.

This clause is invalid if your database is in Automatic Undo Mode, because in that mode Oracle uses undo tablespaces instead of rollback segments.

DEFAULT `DEFAULT` specifies that Oracle will choose automatically which rollback segment to use. If you specify `DEFAULT`, you cannot specify *`rollback_segment`*.

`DEFAULT` is most useful when modifying (rather than creating) a materialized view.

> **See Also:** ALTER MATERIALIZED VIEW on page 9-92

MASTER MASTER specifies the remote rollback segment to be used at the remote master site for the individual materialized view.

LOCAL LOCAL specifies the remote rollback segment to be used for the local refresh group that contains the materialized view.

> **See Also:** *Oracle9i Replication* for information on specifying the local materialized view rollback segment using the DBMS_REFRESH package

If you do not specify MASTER or LOCAL, Oracle uses LOCAL by default. If you do not specify *rollback_segment*, Oracle automatically chooses the rollback segment to be used.

One master rollback segment is stored for each materialized view and is validated during materialized view creation and refresh. If the materialized view is complex, the master rollback segment, if specified, is ignored.

> **See Also:** "Specifying Rollback Segments for Materialized Views: Example" on page 14-30

NEVER REFRESH Clause

Specify NEVER REFRESH to prevent the materialized view from being refreshed with any Oracle refresh mechanism or packaged procedure. Oracle will ignore any REFRESH statement on the materialized view issued from such a procedure. To reverse this clause, you must issue an ALTER MATERIALIZED VIEW ... REFRESH statement.

FOR UPDATE Clause

Specify FOR UPDATE to allow a subquery, primary key, object, or rowid materialized view to be updated. When used in conjunction with Advanced Replication, these updates will be propagated to the master.

> **See Also:** *Oracle9i Replication*

QUERY REWRITE Clause

The QUERY REWRITE clause lets you specify whether the materialized view is eligible to be used for query rewrite.

ENABLE Clause

Specify ENABLE to enable the materialized view for query rewrite.

Restrictions on the ENABLE clause:

- You can enable query rewrite only if all user-defined functions in the materialized view are DETERMINISTIC.
- You can enable query rewrite only if expressions in the statement are repeatable. For example, you cannot include CURRENT_TIME or USER, sequence values (such as the CURRVAL or NEXTVAL pseudocolumns), or the SAMPLE clause (which may sample different rows as the contents of the materialized view change).

Notes:

- Query rewrite is disabled by default, so you must specify this clause to make materialized views eligible for query rewrite.
- Be sure to analyze the materialized view after you create it. Oracle needs the statistics generated by the ANALYZE operation to optimize query rewrite.

See Also:

- *Oracle9i Data Warehousing Guide* for more information on query rewrite
- CREATE FUNCTION on page 13-49

DISABLE Clause

Specify DISABLE to indicate that the materialized view is not eligible for use by query rewrite. However, a disabled materialized view can be refreshed.

AS *subquery*

Specify the defining subquery of the materialized view. When you create the materialized view, Oracle executes this subquery and places the results in the materialized view. This subquery is any valid SQL subquery. However, not all queries are fast refreshable, nor are all subqueries eligible for query rewrite.

Notes on the Defining Query of a Materialized View:

- Oracle does not execute the defining subquery immediately if you specify BUILD DEFERRED.

- Oracle recommends that you qualify each table and view in the FROM clause of the defining subquery of the materialized view with the schema containing it.

  See Also: "AS subquery" clause of CREATE TABLE on page 15-62 for some additional caveats

Restrictions on the Defining Subquery of a Materialized View:

- The defining subquery of a materialized view can select from tables, views, or materialized views owned by the user SYS, but you cannot enable QUERY REWRITE on such a materialized view.
- Materialized join views and materialized aggregate views with a GROUP BY clause cannot select from an index-organized table.
- Materialized views cannot contain columns of datatype LONG.
- You cannot create a materialized view log on a temporary table. Therefore, if the defining subquery references a temporary table, this materialized view will not be eligible for FAST refresh, nor can you specify the QUERY REWRITE clause in this statement.
- If the FROM clause of the defining subquery references another materialized view, then you must always refresh the materialized view referenced in the defining subquery before refreshing the materialized view you are creating in this statement.

If you are creating a materialized view enabled for query rewrite:

- The defining subquery cannot contain (either directly or through a view) references to ROWNUM, USER, SYSDATE, remote tables, sequences, or PL/SQL functions that write or read database or package state.
- Neither the materialized view nor the master tables of the materialized view can be remote.

If you want the materialized view to be eligible for fast refresh using a materialized view log, some additional restrictions may apply.

See Also:

- *Oracle9i Data Warehousing Guide* for more information on restrictions relating to data warehousing
- *Oracle9i Replication* for more information on restrictions relating to replication
- "Creating Materialized Join Views: Example" on page 14-27, "Creating Subquery Materialized Views: Example" on page 14-28 and "Creating a Nested Materialized View: Example" on page 14-31

Examples

The following examples require the materialized logs that are created in the "Examples" section of CREATE MATERIALIZED VIEW on page 14-5.

Creating Materialized Aggregate Views: Example The following statement creates and populates a materialized aggregate view on the sample `sh.sales` table and specifies the default refresh method, mode, and time. It uses the materialized view log created in "Creating a Materialized View Log: Examples" on page 14-39, as well as the two additional logs shown here:

```
CREATE MATERIALIZED VIEW LOG ON times
   WITH ROWID, SEQUENCE (time_id, calendar_year)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW LOG ON products
   WITH ROWID, SEQUENCE (prod_id)
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW sales_mv
   BUILD IMMEDIATE
   REFRESH FAST ON COMMIT
   AS SELECT t.calendar_year, p.prod_id,
      SUM(s.amount_sold) AS sum_sales
      FROM times t, products p, sales s
      WHERE t.time_id = s.time_id AND p.prod_id = s.prod_id
      GROUP BY t.calendar_year, p.prod_id;
```

Creating Materialized Join Views: Example The following statement creates and populates the materialized aggregate view `sales_by_month_by_state` using tables in the sample `sh` schema. The materialized view will be populated with data

as soon as the statement executes successfully. By default, subsequent refreshes will be accomplished by reexecuting the materialized view's query:

```
CREATE MATERIALIZED VIEW sales_by_month_by_state
     TABLESPACE demo
     PARALLEL 4
     BUILD IMMEDIATE
     REFRESH COMPLETE
     ENABLE QUERY REWRITE
     AS SELECT t.calendar_month_desc, c.cust_state_province,
        SUM(s.amount_sold) AS sum_sales
        FROM times t, sales s, customers c
        WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
        GROUP BY t.calendar_month_desc, c.cust_state_province;
```

Creating Prebuilt Materialized Views: Example The following statement creates a materialized aggregate view for the preexisting summary table, `sales_sum_table`:

```
CREATE TABLE sales_sum_table
   (month VARCHAR2(8), state VARCHAR2(40), sales NUMBER(10,2));

CREATE MATERIALIZED VIEW sales_sum_table
   ON PREBUILT TABLE
   ENABLE QUERY REWRITE
   AS SELECT t.calendar_month_desc AS month,
             c.cust_state_province AS state,
             SUM(s.amount_sold) AS sales
      FROM times t, customers c, sales s
      WHERE s.time_id = t.time_id AND s.cust_id = c.cust_id
      GROUP BY t.calendar_month_desc, c.cust_state_province;
```

In this example, the materialized view has the same name as the prebuilt table and also has the same number of columns with the same datatypes as the prebuilt table.

Creating Subquery Materialized Views: Example The following statement creates a subquery materialized view based on the `orders` and `customers` tables in the `sales` schema at a remote database:

```
CREATE MATERIALIZED VIEW sales.orders FOR UPDATE
   AS SELECT * FROM sales.orders@dbs1.acme.com o
   WHERE EXISTS
     (SELECT * FROM sales.customers@dbs1.acme.com c
      WHERE o.c_id = c.c_id);
```

Creating Primary Key Materialized Views: Example The following statement creates the primary-key materialized view `catalog` on the sample table `oe.product_information`:

```
CREATE MATERIALIZED VIEW catalog
   REFRESH FAST START WITH SYSDATE NEXT  SYSDATE + 1/4096
   WITH PRIMARY KEY
   AS SELECT * FROM product_information;
```

Creating Rowid Materialized Views: Example The following statement creates a rowid materialized view on the sample table `oe.orders`:

```
CREATE MATERIALIZED VIEW order_data REFRESH WITH ROWID
   AS SELECT * FROM orders;
```

Periodic Refresh of Materialized Views: Example The following statement creates the primary key materialized view `emp_data` and populates it with data from the sample table `hr.employees`:

```
CREATE MATERIALIZED VIEW LOG ON employees
   WITH PRIMARY KEY
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW emp_data
   PCTFREE 5 PCTUSED 60
   TABLESPACE demo
   STORAGE (INITIAL 50K NEXT 50K)
   REFRESH FAST NEXT sysdate + 7
   AS SELECT * FROM employees;
```

The statement does not include a `START WITH` parameter, so Oracle determines the first automatic refresh time by evaluating the `NEXT` value using the current `SYSDATE`. A materialized view log was created for the employee table, so Oracle performs a fast refresh of the materialized view every 7 days, beginning 7 days after the materialized view is created.

Because the materialized view conforms to the conditions for fast refresh, Oracle will perform a fast refresh. The preceding statement also establishes storage characteristics that Oracle uses to maintain the materialized view.

Automatic Refresh Times for Materialized Views: Example The following statement creates the complex materialized view `all_emps` that queries the employee tables in Dallas and Baltimore:

```
CREATE MATERIALIZED VIEW all_emps
```

```
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE (INITIAL 50K NEXT 50K)
   USING INDEX STORAGE (INITIAL 25K NEXT 25K)
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE), 'MONDAY')) + 15/24
   AS SELECT * FROM fran.emp@dallas
         UNION
      SELECT * FROM marco.emp@balt;
```

Oracle automatically refreshes this materialized view tomorrow at 11:00 a.m. and subsequently every Monday at 3:00 p.m.. The default refresh method is FORCE. all_emps contains a UNION operator, which is not supported for fast refresh, so Oracle will automatically perform a complete refresh.

The preceding statement also establishes storage characteristics for both the materialized view and the index that Oracle uses to maintain it:

- The first establishes the sizes of the first and second extents of the materialized view as 50 kilobytes each.
- The second (appearing with the USING INDEX clause) establishes the sizes of the first and second extents of the index as 25 kilobytes each.

Specifying Rollback Segments for Materialized Views: Example The following statement creates the primary key materialized view sales_emp with rollback segment master_seg at the remote master and rollback segment snap_seg for the local refresh group that contains the materialized view.

> **Note:** This example is not relevant if your database is in Automatic Undo Mode, because in that mode Oracle uses undo tablespaces instead of rollback segments.

```
CREATE MATERIALIZED VIEW sales_emp
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
   USING MASTER ROLLBACK SEGMENT master_seg
   LOCAL ROLLBACK SEGMENT snap_seg
   AS SELECT * FROM bar;
```

The following statement is incorrect and generates an error because it specifies a segment name with a DEFAULT rollback segment:

```
/* The following statement is invalid. */
```

```
CREATE MATERIALIZED VIEW order_mv
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
   USING DEFAULT ROLLBACK SEGMENT mv_seg
   AS SELECT * FROM orders;
```

Creating a Fast Refreshable Materialized View: Example The following statement creates a fast-refreshable materialized view that selects columns from the `order_items` table in the sample `oe` schema, using the `UNION` set operator to restrict the rows returned from the `product_information` and `inventories` tables using `WHERE` conditions. The materialized view logs for `order_items` and `product_information` were created in the "Examples" section of `CREATE MATERIALIZED VIEW LOG` on page 14-39. This example requires a materialized view log on `oe.inventories`.

```
CREATE MATERIALIZED VIEW LOG ON inventories
   WITH (quantity_on_hand);

CREATE MATERIALIZED VIEW warranty_orders REFRESH FAST AS
  SELECT order_id, line_item_id, product_id FROM order_items o
    WHERE EXISTS
    (SELECT * FROM inventories i WHERE o.product_id = i.product_id
      AND i.quantity_on_hand IS NOT NULL)
  UNION
    SELECT order_id, line_item_id, product_id FROM order_items
    WHERE quantity > 5;
```

This materialized view requires that materialized view logs be defined on `order_items` (with `product_id` as a join column) and on inventories (with `quantity_on_hand` as a filter column). See "Specifying Filter Columns for Materialized View Logs: Example" and "Specifying Join Columns for Materialized View Logs: Example" on page 14-40.

Creating a Nested Materialized View: Example The following example uses the materialized view from the preceding example as a master table to create a materialized view tailored for a particular sales representative in the sample `oe` schema:

```
CREATE MATERIALIZED VIEW my_warranty_orders
   AS SELECT w.order_id, w.line_item_id, o.order_date
   FROM warranty_orders w, orders o
   WHERE o.order_id = o.order_id
   AND o.sales_rep_id = 165;
```

CREATE MATERIALIZED VIEW LOG

Purpose

Use the CREATE MATERIALIZED VIEW LOG statement to create a **materialized view log**, which is a table associated with the master table of a materialized view.

Note: The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

When DML changes are made to the master table's data, Oracle stores rows describing those changes in the materialized view log and then uses the materialized view log to refresh materialized views based on the master table. This process is called an incremental or **fast refresh**. Without a materialized view log, Oracle must reexecute the materialized view query to refresh the materialized view. This process is called a **complete refresh**. Usually, a fast refresh takes less time than a complete refresh.

A materialized view log is located in the master database in the same schema as the master table. A master table can have only one materialized view log defined on it. Oracle can use this materialized view log to perform fast refreshes for all fast-refreshable materialized views based on the master table.

To fast refresh a materialized join view (a materialized view containing a join), you must create a materialized view log for each of the tables referenced by the materialized view.

See Also:

- CREATE MATERIALIZED VIEW on page 14-5, ALTER MATERIALIZED VIEW on page 9-92, *Oracle9i Database Concepts*, *Oracle9i Data Warehousing Guide*, and *Oracle9i Replication* for information on materialized views in general
- ALTER MATERIALIZED VIEW LOG on page 9-112 for information on modifying a materialized view log
- DROP MATERIALIZED VIEW LOG on page 16-85 for information on dropping a materialized view log
- *Oracle9i Database Concepts* and *Oracle9i Database Utilities* for information on using direct loader logs

Prerequisites

The privileges required to create a materialized view log directly relate to the privileges necessary to create the underlying objects associated with a materialized view log.

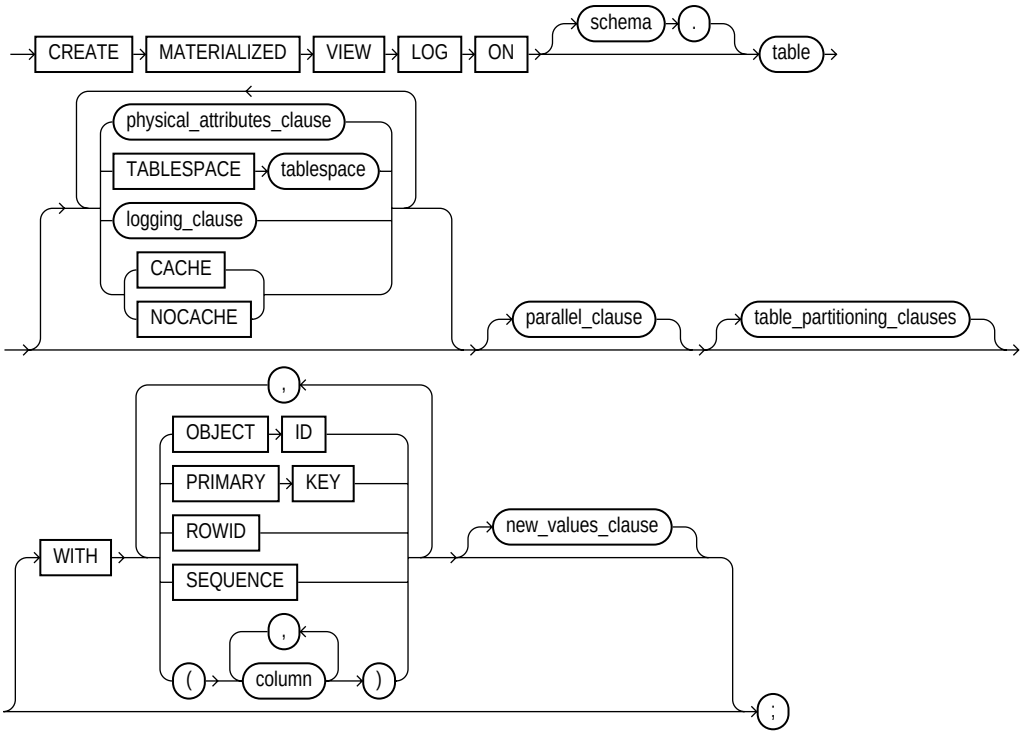
- If you own the master table, you can create an associated materialized view log if you have the CREATE TABLE privilege.
- If you are creating a materialized view log for a table in another user's schema, you must have the CREATE ANY TABLE and COMMENT ANY TABLE privileges, as well as either the SELECT privilege for the master table or SELECT ANY TABLE.

In either case, the owner of the materialized view log must have sufficient quota in the tablespace intended to hold the materialized view log or must have the UNLIMITED TABLESPACE system privilege.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information about the prerequisites for creating a materialized view log

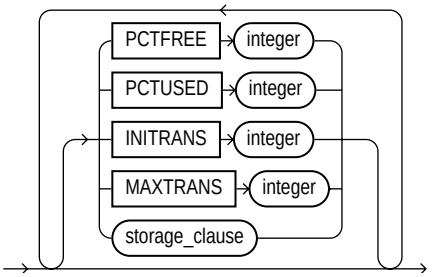
Syntax

create_materialized_vw_log::=



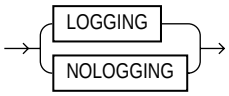
(*physical_attributes_clause::=* on page 14-11, *logging_clause::=* on page 14-35, *parallel_clause::=* on page 14-35, *table_partitioning_clauses* on page 15-44—part of CREATE TABLE syntax, *new_values_clause::=* on page 14-35)

physical_attributes_clause::=

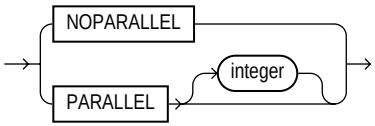


(*storage_clause* on page 7-56)

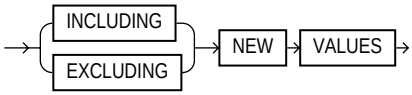
logging_clause::=



parallel_clause::=



new_values_clause::=



Keywords and Parameters

schema

Specify the schema containing the materialized view log's master table. If you omit *schema*, Oracle assumes the master table is contained in your own schema. Oracle creates the materialized view log in the schema of its master table. You cannot create a materialized view log for a table in the schema of the user SYS.

table

Specify the name of the master table for which the materialized view log is to be created.

Restriction on *table***:** You cannot create a materialized view log for a temporary table or for a view.

> **See Also:** "Creating a Materialized View Log: Examples" on page 14-39

physical_attributes_clause

Use the *physical_attributes_clause* to establish values for physical and storage characteristics for the materialized view log.

See Also:

- *physical_attributes_clause* on page 7-52 for a complete description of the parameters of the *physical_attributes_clause*
- *storage_clause* on page 7-56 for a complete description of the *storage_clause*, including default values

TABLESPACE Clause

Specify the tablespace in which the materialized view log is to be created. If you omit this clause, Oracle creates the materialized view log in the default tablespace of the schema of the materialized view log.

logging_clause

Specify either LOGGING or NOLOGGING to establish the logging characteristics for the materialized view log. The default is the logging characteristic of the tablespace in which the materialized view log resides.

See Also: *logging_clause* on page 7-45 for a full description of this clause

CACHE | NOCACHE

For data that will be accessed frequently, CACHE specifies that the blocks retrieved for this log are placed at the *most recently used* end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

NOCACHE specifies that the blocks are placed at the *least recently used* end of the LRU list. The default is NOCACHE.

Note: NOCACHE has no effect on materialized view logs for which you specify KEEP in the *storage_clause*.

See Also: CREATE TABLE on page 15-7 for information about specifying CACHE or NOCACHE

parallel_clause

The *parallel_clause* lets you indicate whether parallel operations will be supported for the materialized view log.

Note: The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

> **See Also:** "Notes on the parallel_clause" for CREATE TABLE on page 15-56

table_partitioning_clauses

Use the *table_partitioning_clauses* to indicate that the materialized view log is partitioned on specified ranges of values or on a hash function. Partitioning of materialized view logs is the same as partitioning of tables.

> **See Also:** *table_partitioning_clauses* of CREATE TABLE on page 15-44

WITH Clause

Use the WITH clause to indicate whether the materialized view log should record the primary key, the rowid, object ID, or a combination of these row identifiers when rows in the master are changed. You can also use this clause to add a sequence to the materialized view log to provide additional ordering information for its records.

This clause also specifies whether the materialized view log records additional columns that might be referenced as **filter columns** (non-primary-key columns referenced by subquery materialized views) or **join columns** (non-primary-key columns that define a join in the subquery WHERE clause).

If you omit this clause, or if you specify the clause without PRIMARY KEY, ROWID, or OBJECT ID, then Oracle stores primary key values by default. However, Oracle does not store primary key values implicitly if you specify only OBJECT ID or ROWID at create time. A primary key log, created either explicitly or by default, performs additional checking on the primary key constraint.

OBJECT ID Specify OBJECT ID to indicate that the system-generated or user-defined object identifier of every modified row should be recorded in the materialized view log.

Restriction on OBJECT ID: You can specify OBJECT ID only when creating a log on an object table, and you cannot specify it for storage tables.

PRIMARY KEY Specify PRIMARY KEY to indicate that the primary key of all rows changed should be recorded in the materialized view log.

ROWID Specify ROWID to indicate that the rowid of all rows changed should be recorded in the materialized view log.

SEQUENCE Specify SEQUENCE to indicate that a sequence value providing additional ordering information should be recorded in the materialized view log. Sequence numbers are necessary to support fast refresh after some update scenarios.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information on the use of sequence numbers in materialized view logs and for examples that use this clause

column Specify the columns whose values you want to be recorded in the materialized view log for all rows that are changed. Typically these columns are filter columns (non-primary-key columns referenced by materialized views) and join columns (non-primary-key columns that define a join in the WHERE clause of the subquery).

Restrictions on the WITH Clause:

- You can specify only one PRIMARY KEY, one ROWID, one OBJECT ID, and one column list for each materialized view log.
- Primary key columns are implicitly recorded in the materialized view log. Therefore, you cannot specify either of the following combinations if *column* contains one of the primary key columns:

  ```
  WITH ... PRIMARY KEY ... (column)
  ```

```
WITH ... (column) ... PRIMARY KEY
WITH (column)
```

See Also:

- CREATE MATERIALIZED VIEW on page 14-5 for information on explicit and implicit inclusion of materialized view log values
- *Oracle9i Replication* for more information about filter columns and join columns
- "Specifying Filter Columns for Materialized View Logs: Example" on page 14-40 and "Specifying Join Columns for Materialized View Logs: Example" on page 14-40

NEW VALUES Clause

The NEW VALUES clause lets you indicate whether Oracle saves both old and new values in the materialized view log.

See Also: "Including New Values in Materialized View Logs: Example" on page 14-40

INCLUDING Specify INCLUDING to save both new and old values in the log. If this log is for a table on which you have a single-table materialized aggregate view, and if you want the materialized view to be eligible for fast refresh, you must specify INCLUDING.

EXCLUDING Specify EXCLUDING to disable the recording of new values in the log. This is the default. You can use this clause to avoid the overhead of recording new values. However, do not use this clause if you have a fast-refreshable single-table materialized aggregate view defined on this table.

Examples

Creating a Materialized View Log: Examples The following statement creates a materialized view log on the oe.customers table that specifies physical and storage characteristics:

```
CREATE MATERIALIZED VIEW LOG ON customers
   PCTFREE 5
   TABLESPACE demo
   STORAGE (INITIAL 10K NEXT 10K);
```

This materialized view log supports fast refresh for primary key materialized views only. The following statement creates another version of the materialized view log with the ROWID clause, which enables fast refresh for more types of materialized views:

```
CREATE MATERIALIZED VIEW LOG ON customers WITH PRIMARY KEY, ROWID;
```

This materialized view log makes fast refresh possible for rowid materialized views and for materialized join views. To provide for fast refresh of materialized aggregate views, you must also specify the SEQUENCE and INCLUDING NEW VALUES clauses, as shown in the next statement.

Specifying Filter Columns for Materialized View Logs: Example The following statement creates a materialized view log on the sh.sales table, and is used in "Creating Materialized Aggregate Views: Example" on page 14-27. It specifies as filter columns all of the columns of the table referenced in that materialized view.

```
CREATE MATERIALIZED VIEW LOG ON sales
   WITH ROWID, SEQUENCE(amount_sold, time_id, prod_id)
   INCLUDNG NEW VALUES;
```

Specifying Join Columns for Materialized View Logs: Example The following statement creates a materialized view log on the order_items table of the sample oe schema. The log records primary keys and product_id, which is used as a join column in "Creating a Fast Refreshable Materialized View: Example" on page 14-31.

```
CREATE MATERIALIZED VIEW LOG ON order_items WITH (product_id);
```

Including New Values in Materialized View Logs: Example The following example creates a materialized view log on the oe.product_information table that specifies INCLUDING NEW VALUES:

```
CREATE MATERIALIZED VIEW LOG ON product_information
   WITH ROWID, (list_price, min_price, category_id)
   INCLUDING NEW VALUES;
```

You could create the following materialized aggregate view to use the product_information log:

```
CREATE MATERIALIZED VIEW products_mv
   REFRESH FAST ON COMMIT
   AS SELECT SUM(list_price - min_price), category_id
         FROM product_information
         GROUP BY category_id;
```

This materialized view is eligible for fast refresh because the log it uses includes both old and new values.

CREATE OPERATOR

Purpose

Use the CREATE OPERATOR statement to create a new operator and define its bindings.

Operators can be referenced by indextypes and by DML and query SQL statements. The operators, in turn, reference functions, packages, types, and other user-defined objects.

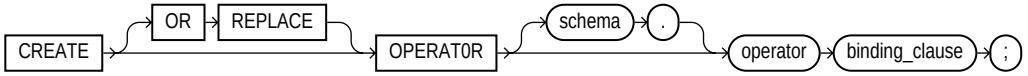
> **See Also:** *Oracle9i Data Cartridge Developer's Guide* and *Oracle9i Database Concepts* for a discussion of these dependencies and of operators in general

Prerequisites

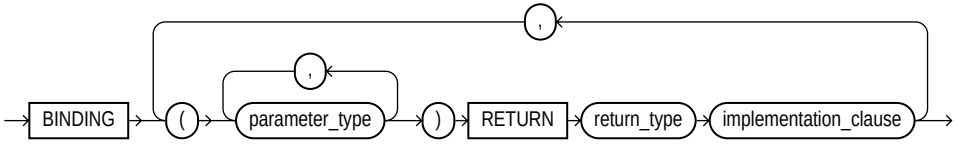
To create an operator in your own schema, you must have CREATE OPERATOR system privilege. To create an operator in another schema, you must have the CREATE ANY OPERATOR system privilege. In either case, you must also have EXECUTE privilege on the functions and operators referenced.

Syntax

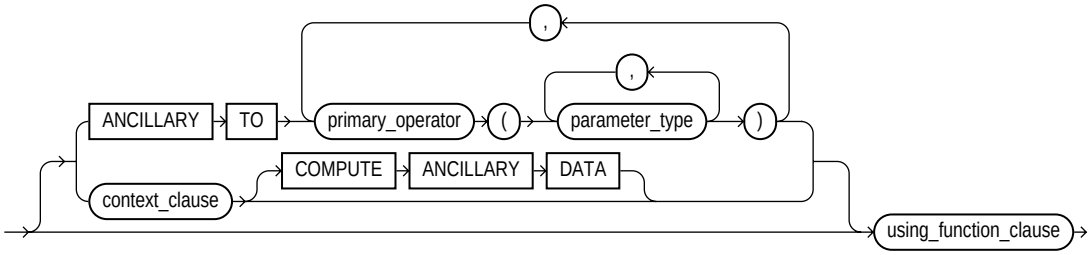
create_operator::=



binding_clause::=

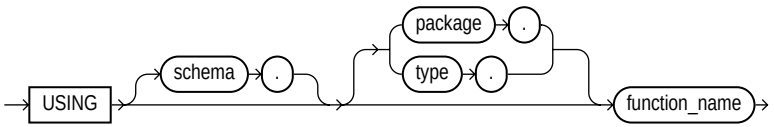


implementation_clause::=



context_clause::=

using_function_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to replace the definition of the operator schema object.

Restriction on replacing an operator: You can replace the definition only if the operator has no dependent objects (for example, indextypes supporting the operator).

schema

Specify the schema containing the operator. If you omit *schema*, Oracle creates the operator in your own schema.

operator

Specify the name of the operator to be created.

binding_clause

Use the *binding_clause* to specify one or more parameter datatypes (*parameter_type*) for binding the operator to a function. The signature of each binding (that is, the sequence of the datatypes of the arguments to the corresponding function) must be unique according to the rules of overloading.

The *parameter_type* can itself be an object type. If it is, you can optionally qualify it with its schema.

Restriction on the *binding_clause***:** You cannot specify a *parameter_type* of REF, LONG, or LONG RAW.

> **See Also:** *PL/SQL User's Guide and Reference* for more information about overloading

RETURN Clause

Specify the return datatype for the binding.

The *return_type* can itself be an object type. If so, you can optionally qualify it with its schema.

Restriction on the RETURN clause: You cannot specify a *return_type* of REF, LONG, or LONG RAW.

implementation_clause

ANCILLARY TO Clause

Use the ANCILLARY TO clause to indicate that the operator binding is ancillary to the specified primary operator binding (*primary_operator*). If you specify this clause, do not specify a previous binding with just one number parameter.

context_clause

Use the *context_clause* to specify the name of the implementation type used by the functional implementation of the operator as a scan context.

COMPUTE ANCILLARY DATA Specify COMPUTE ANCILLARY DATA to indicate that the operator binding computes ancillary data.

using_function_clause

The *using_function_clause* lets you specify the function that provides the implementation for the binding. *function_name* can be a standalone function, packaged function, type method, or a synonym for any of these.

Example

Creating User-Defined Operators: Example This example creates a very simple functional implementation of equality and then creates an operator that uses the function:

```
CREATE FUNCTION eq_f(a VARCHAR2, b VARCHAR2) RETURN NUMBER AS
BEGIN
   IF a = b THEN RETURN 1;
   ELSE RETURN 0;
   END IF;
END;
/

CREATE OPERATOR eq_op
   BINDING (VARCHAR2, VARCHAR2)
   RETURN NUMBER
   USING eq_f;
```

CREATE OUTLINE

Purpose

Use the CREATE OUTLINE statement to create a **stored outline**, which is a set of attributes used by the optimizer to generate an execution plan. You can then instruct the optimizer to use a set of outlines to influence the generation of execution plans whenever a particular SQL statement is issued, regardless of changes in factors that can affect optimization. You can also modify an outline so that it takes into account changes in these factors.

Note: The SQL statement issued subsequently must be an exact string match of the statement specified when creating the outline.

See Also:

- *Oracle9i Database Performance Tuning Guide and Reference* for information on execution plans
- ALTER OUTLINE on page 9-120 for information on modifying an outline
- ALTER SESSION on page 10-2 and ALTER SYSTEM on page 10-22 for information on the USE_STORED_OUTLINES and USE_PRIVATE_OUTLINES parameters

Prerequisites

To create a public or private outline, you must have the CREATE ANY OUTLINE system privilege.

If you are creating a clone outline from a source outline, you must also have the SELECT_CATALOG_ROLE role.

To create a private outline, you must provide an outline editing table to hold the outline data in your schema by executing the DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES procedure. You must have the EXECUTE privilege on the DBMS_OUTLN_EDIT package to execute this procedure.

You enable or disable the use of stored outlines dynamically for an individual session or for the system:

- Enable the USE_STORED_OUTLINES parameter to use public outlines

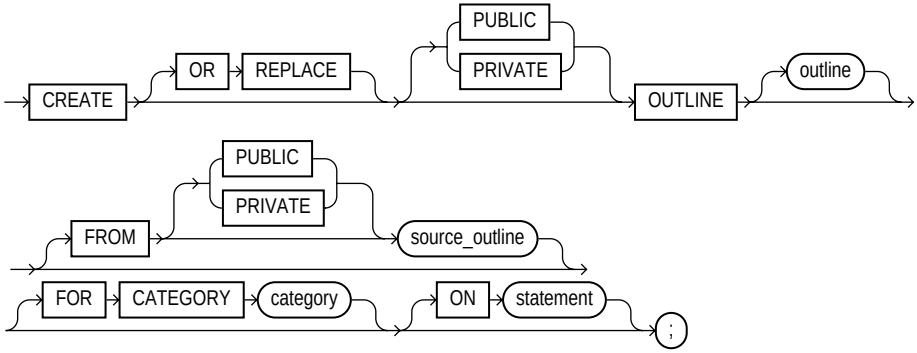
- Enable the USE_PRIVATE_OUTLINES parameter to use private stored outlines.

See Also:

- *Oracle9i Database Performance Tuning Guide and Reference* for information on using outlines for performance tuning
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the DBMS_OUTLN_EDIT package

Syntax

create_outline::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to replace an existing outline with a new outline of the same name.

PUBLIC | PRIVATE

Specify PUBLIC if you are creating an outline for use by PUBLIC. This is the default.

Specify PRIVATE to create an outline for private use by the current session only. The data of this outline is stored in the current schema.

Note: Before first creating a private outline, you must run the OUTLN_PKG.CREATE_EDIT_TABLES procedure to create the required outline tables and indexes in your schema.

---

outline

Specify the unique name to be assigned to the stored outline. If you do not specify *outline*, the system generates an outline name.

> **See Also:** "Creating an Outline: Example" on page 14-49

FROM ... *source_outline* Clause

Use the FROM clause to create a new outline by copying an existing one. By default, Oracle looks for *source_category* in the public area. If you specify PRIVATE, Oracle will look for the outline in the current schema.

Restriction on the FROM clause: If you specify the FROM clause, you cannot specify the ON clause.

> **See Also:** "Creating a Private Clone Outline: Example" on page 14-49 and "Publicizing a Private Outline to the Public Area: Example" on page 14-49

FOR CATEGORY Clause

Specify an optional name used to group stored outlines. For example, you could specify a category of outlines for end-of-week use and another for end-of-quarter use. If you do not specify *category*, the outline is stored in the DEFAULT category.

ON Clause

Specify the SQL statement for which Oracle will create an outline when the statement is compiled. This clause is optional only if you are creating a copy of an existing outline using the FROM clause.

You can specify any one of the following statements:

- SELECT
- DELETE
- UPDATE
- INSERT ... SELECT
- CREATE TABLE ... AS SELECT

Restrictions on the ON clause:

- If you specify the ON clause, you cannot specify the FROM clause.
- You cannot create an outline on a multitable INSERT statement.

Note: You can specify multiple outlines for a single statement, but each outline for the same statement must be in a different category.

Example

Creating an Outline: Example The following statement creates a stored outline by compiling the ON statement. The outline is called salaries and is stored in the category special.

```
CREATE OUTLINE salaries FOR CATEGORY special
   ON SELECT last_name, salary FROM employees;
```

When this same SELECT statement is subsequently compiled, if the USE_STORED_OUTLINES parameter is set to special, Oracle generates the same execution plan as was generated when the outline salaries was created.

Creating a Private Clone Outline: Example The following statement creates a stored private outline my_salaries based on the public category salaries created in the preceding example. In order to create a private outline, the user creating the private outline must have the EXECUTE privilege on the DBMS_OUTLN_EDIT package, and must execute the CREATE_EDIT_TABLES procedure of that package.

```
EXECUTE DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES;

CREATE OR REPLACE PRIVATE OUTLINE my_salaries
   FROM salaries;
```

Publicizing a Private Outline to the Public Area: Example The following statement copies back (or publicizes) a private outline to the public area after private editing:

```
CREATE OR REPLACE OUTLINE public_salaries
   FROM PRIVATE my_salaries;
```

CREATE PACKAGE

Purpose

Use the CREATE PACKAGE statement to create the specification for a stored **package**, which is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The **specification** declares these objects.

See Also:

- CREATE PACKAGE BODY on page 14-55 for information on specifying the implementation of the package
- CREATE FUNCTION on page 13-49 and CREATE PROCEDURE on page 14-62 for information on creating standalone functions and procedures
- ALTER PACKAGE on page 9-122 for information on modifying a package
- DROP PACKAGE on page 16-90 for information on dropping a package
- *Oracle9i Application Developer's Guide - Fundamentals* and *Oracle9i Supplied PL/SQL Packages and Types Reference* for detailed discussions of packages and how to use them

Prerequisites

Before a package can be created, the user SYS must run a SQL script commonly called DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

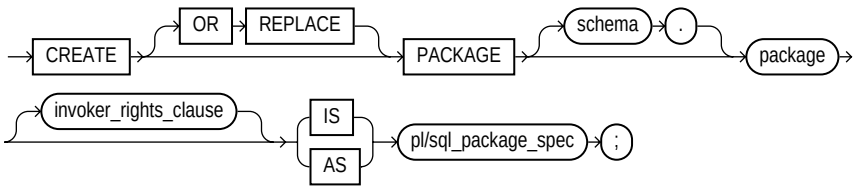
To create a package in your own schema, you must have the CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have the CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

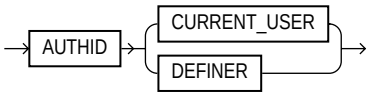
See Also: *PL/SQL User's Guide and Reference*

Syntax

create_package::=



invoker_rights_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the package specification if it already exists. Use this clause to change the specification of an existing package without dropping, re-creating, and regranting object privileges previously granted on the package. If you change a package specification, Oracle recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, Oracle marks the indexes DISABLED.

> **See Also:** ALTER PACKAGE on page 9-122 for information on recompiling package specifications

schema

Specify the schema to contain the package. If you omit *schema*, Oracle creates the package in your own schema.

package

Specify the name of the package to be created.

If creating the package results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*PLUS command SHOW ERRORS.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the functions and procedures in the package execute with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding package body as well.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the package.

AUTHID CURRENT_USER

Specify CURRENT_USER to indicate that the package executes with the privileges of CURRENT_USER. This clause creates an **invoker-rights package**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the package resides.

AUTHID DEFINER

Specify DEFINER to indicate that the package executes with the privileges of the owner of the schema in which the package resides and that external names resolve in the schema where the package resides. This is the default and creates a **definer-rights package**.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined

pl/sql_package_spec

Specify the package specification, which can contain type definitions, cursor declarations, variable declarations, constant declarations, exception declarations, PL/SQL subprogram specifications, and call specifications (declarations of a C or Java routine expressed in PL/SQL).

See Also:

- *PL/SQL User's Guide and Reference* for more information on PL/SQL package program units
- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on Oracle supplied packages
- "Restrictions on user-defined functions:" on page 13-53 for a list of restrictions on user-defined functions in a package

Example

Creating a Package: Example The following SQL statement creates the specification of the `emp_mgmt` package (PL/SQL is show in italics):

```
CREATE PACKAGE emp_mgmt AS
   FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2,
      manager_id NUMBER, salary NUMBER,
      commission_pct NUMBER, department_id NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(department_id NUMBER, location NUMBER)
      RETURN NUMBER;
   PROCEDURE remove_emp(employee_id NUMBER);
   PROCEDURE remove_dept(department_id NUMBER);
   PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER);
   PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER);
   no_comm EXCEPTION;
   no_sal EXCEPTION;
END emp_mgmt;
/
```

The specification for the `emp_mgmt` package declares the following public program objects:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`
- The exceptions `no_comm` and `no_sal`

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that call any of the package's public procedures or functions or raise any of the package's public exceptions.

Before you can call this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a CREATE PACKAGE BODY statement that creates the body of the `emp_mgmt` package, see CREATE PACKAGE BODY on page 14-55.

CREATE PACKAGE BODY

Purpose

Use the CREATE PACKAGE BODY statement to create the body of a stored **package**, which is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The **body** defines these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

See Also:

- CREATE FUNCTION on page 13-49 and CREATE PROCEDURE on page 14-62 for information on creating standalone functions and procedures
- CREATE PACKAGE on page 14-50 for a discussion of packages, including how to create packages
- "Examples" on page 14-57 for some illustrations
- ALTER PACKAGE on page 9-122 for information on modifying a package
- DROP PACKAGE on page 16-90 for information on removing a package from the database

Prerequisites

Before a package can be created, the user SYS must run a SQL script commonly called DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

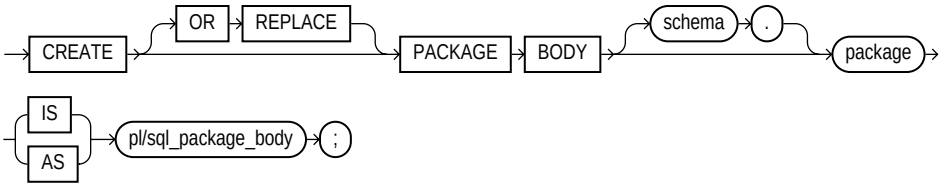
To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE BODY statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

See Also: *PL/SQL User's Guide and Reference*

Syntax

create_package_body::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the package body if it already exists. Use this clause to change the body of an existing package without dropping, re-creating, and regranting object privileges previously granted on it. If you change a package body, Oracle recompiles it.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

> **See Also:** ALTER PACKAGE on page 9-122 for information on recompiling package bodies

schema

Specify the schema to contain the package. If you omit *schema*, Oracle creates the package in your current schema.

package

Specify the name of the package to be created.

pl/sql_package_body

Specify the package body, which can contain PL/SQL subprogram bodies or call specifications (declarations of a C or Java routine expressed in PL/SQL).

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information on writing PL/SQL or C package program units
- *Oracle9i Java Stored Procedures Developer's Guide* for information on Java package program units
- "Restrictions on user-defined functions:" on page 13-53 for a list of restrictions on user-defined functions in a package

Examples

Creating a Package Body: Example This SQL statement creates the body of the `emp_mgmt` package created in "Creating a Package: Example" on page 14-53 (PL/SQL is shown in italics):

```
CREATE PACKAGE BODY emp_mgmt AS
   tot_emps NUMBER;
   tot_depts NUMBER;

FUNCTION hire
   (last_name VARCHAR2, job_id VARCHAR2,
   manager_id NUMBER, salary NUMBER,
   commission_pct NUMBER, department_id NUMBER
RETURN NUMBER IS
   new_empno NUMBER;
BEGIN
   SELECT employees_seq.NEXTVAL
      INTO new_empno
      FROM DUAL;
   INSERT INTO employees
      VALUES (new_empno, last_name, job_id, manager_id, salary,
      commission_pct, department_id, tot_emps := tot_emps + 1);
   RETURN(new_empno);
END;

FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
   RETURN NUMBER IS
      new_deptno NUMBER;
   BEGIN
      SELECT departments_seq.NEXTVAL
         INTO new_deptno
         FROM dual;
      INSERT INTO departments
```

```
         VALUES (new_deptno, department_id, location_id);
            tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;

PROCEDURE remove_emp (employee_id NUMBER) IS
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
         tot_emps := tot_emps - 1;
   END;

PROCEDURE remove_dept(department_id NUMBER) IS
   BEGIN
      DELETE FROM departments
      WHERE departments.department_id = remove_dept.department_id;
         tot_depts := tot_depts - 1;
      SELECT COUNT(*) INTO tot_emps FROM employees;
   /* In case, oracle deleted employees from the EMPLOYEES
   table to enforce referential integrity constraints, reset
   the value of the variable TOT_EMPS to the total number of
   employees in the EMPLOYEES table. */
   END;

PROCEDURE increase_sal (employee_id NUMBER, sal_incr NUMBER) IS
   curr_sal NUMBER;
   BEGIN
      SELECT salary INTO curr_sal FROM employees
      WHERE employees.employee_id = increase_sal.employee_id;
      IF curr-Sal IS NULL
         THEN RAISE no_sal;
      ELSE
         UPDATE employees
         SET salary = salary + sal_incr
         WHERE employee_id = employee_id;
      ENDIF;
   END;

PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER) IS
      curr_comm NUMBER;
   BEGIN
      SELECT commission_pct
      INTO curr_comm
      FROM employees
      WHERE employees.employee_id = increase_comm.employee_id
```

```
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE employees
         SET commission_pct = commission_pct + comm_incr;
      END IF;
   END;

END emp_mgmt;
/
```

The package body defines the public program objects declared in the package specification:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`

These objects are declared in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure `increase_all_comms` separate from the `emp_mgmt` package that calls the `increase_comm` procedure.

These objects are defined in the package body, so you can change their definitions without causing Oracle to invalidate dependent schema objects. For example, if you subsequently change the definition of `hire`, Oracle need not recompile `increase_all_comms` before executing it.

The package body in this example also declares private program objects, the variables `tot_emps` and `tot_depts`. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable `tot_depts`. However, the function `create_dept` is part of the package, so `create_dept` can change the value of tot_depts.

CREATE PFILE

Purpose

Use the CREATE PFILE statement to export a binary server parameter file into a text initialization parameter file. Creating a text parameter file is a convenient way to get a listing of the current parameter settings being used by the database, and it lets you edit the file easily in a text editor and then convert it back into a server parameter file using the CREATE SPFILE statement.

Upon successful execution of this statement, Oracle creates a text parameter file on the server. In a Real Application Clusters environment, it will contain all parameter settings of all instances. It will also contain any comments that appeared on the same line with a parameter setting in the server parameter file.

See Also:

- CREATE SPFILE on page 14-92 for information on server parameter files
- *Oracle9i Database Administrator's Guide* for information on pre-Oracle9*i* text initialization parameter files and Oracle9*i* binary server parameter files
- *Oracle9i Real Application Clusters Administration* for information on using server parameter files in a Real Application Clusters environment

Prerequisites

You must have the SYSDBA or the SYSOPER role to execute this statement. You can execute this statement either before or after instance startup.

Syntax

create_pfile::=

Keywords and Parameters

pfile_name

Specify the name of the text parameter file you want to create. If you do not specify *pfile_name*, Oracle uses the platform-specific default initialization parameter file name.

spfile_name

Specify the name of the binary server parameter from which you want to create a text file.

- If you specify *spfile_name*, the file must exist on the server. If the file does not reside in the default directory for server parameter files on your operating system, you must specify the full path.
- If you do not specify *spfile_name*, Oracle looks in the default directory for server parameter files on your operating system, for the platform-specific default server parameter file name, and uses that file. If that file does not exist in the expected directory, Oracle returns an error.

> **See Also:** *Oracle9i Database Administrator's Guide for Windows* (or other appropriate operating system specific documentation) for default parameter file names

Examples

Creating a Parameter File: Example The following example creates a text parameter file `my_init.ora` from a binary server parameter file `production.ora`:

```
CREATE PFILE = 'my_init.ora' FROM SPFILE = 's_params.ora';
```

> **Note:** Typically you will need to specify the full path and filename for parameter files on your operating system. Please refer to your Oracle operating system documentation for path information.

CREATE PROCEDURE

Purpose

Use the CREATE PROCEDURE statement to create a standalone stored procedure or a call specification.

A **procedure** is a group of PL/SQL statements that you can call by name. A **call specification ("call spec")** declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells Oracle which Java method to invoke when a call is made. It also tells Oracle what type conversions to make for the arguments and return value.

Stored procedures offer advantages in the areas of development, integrity, security, performance, and memory allocation.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information on stored procedures, including how to call stored procedures and for information about registering external procedures
- CREATE FUNCTION on page 13-49 for information specific to functions, which are similar to procedures in many ways
- CREATE PACKAGE on page 14-50 for information on creating packages. (The CREATE PROCEDURE statement creates a procedure as a standalone schema object. You can also create a procedure as part of a package.)
- ALTER PROCEDURE on page 9-126 and DROP PROCEDURE on page 16-92 for information on modifying and dropping a standalone procedure
- CREATE LIBRARY on page 14-2 for more information about shared libraries

Prerequisites

Before creating a procedure, the user SYS must run a SQL script commonly called DBMSSTDX.SQL. The exact name and location of this script depends on your operating system.

To create a procedure in your own schema, you must have the CREATE PROCEDURE system privilege. To create a procedure in another user's schema, you must have

CREATE ANY PROCEDURE system privilege. To replace a procedure in another schema, you must have the ALTER ANY PROCEDURE system privilege.

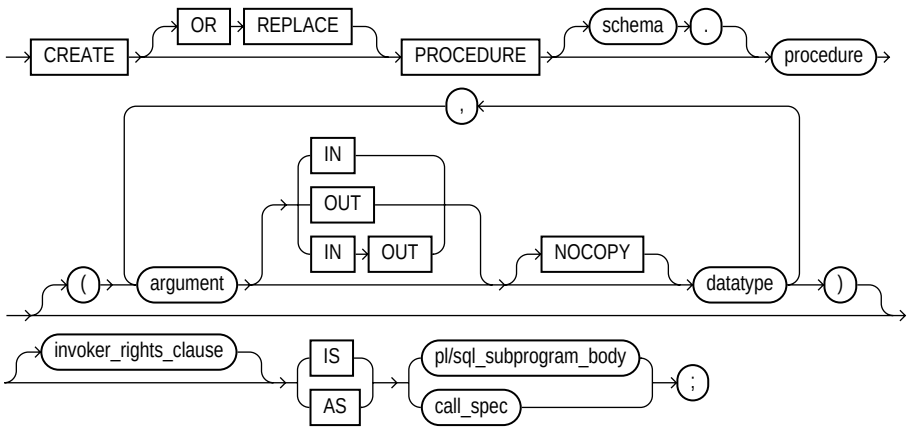
To invoke a call spec, you may need additional privileges (for example, EXECUTE privileges on the C library for a C call spec).

To embed a CREATE PROCEDURE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

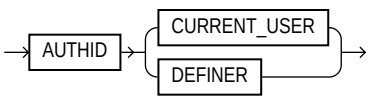
See Also: *PL/SQL User's Guide and Reference* or *Oracle9i Java Stored Procedures Developer's Guide* for more information

Syntax

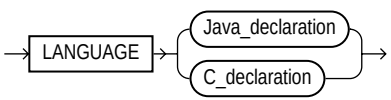
create_procedure::=



invoker_rights_clause::=

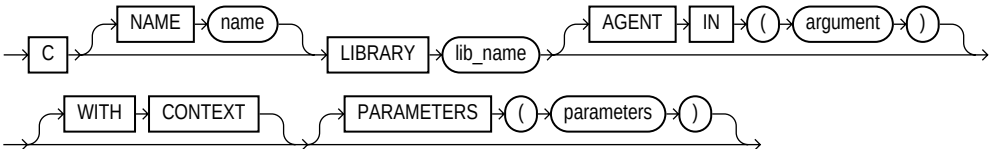


call_spec::=



Java_declaration::=

C_declaration::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping, re-creating, and regranting object privileges previously granted on it. If you redefine a procedure, Oracle recompiles it.

Users who had previously been granted privileges on a redefined procedure can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the package, Oracle marks the indexes DISABLED.

> **See Also:** ALTER PROCEDURE on page 9-126 for information on recompiling procedures

schema

Specify the schema to contain the procedure. If you omit *schema*, Oracle creates the procedure in your current schema.

procedure

Specify the name of the procedure to be created.

If creating the procedure results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

argument

Specify the name of an argument to the procedure. If the procedure does not accept arguments, you can omit the parentheses following the procedure name.

IN Specify IN to indicate that you must specify a value for the argument when calling the procedure.

OUT Specify OUT to indicate that the procedure passes a value for this argument back to its calling environment after execution.

IN OUT Specify IN OUT to indicate that you must specify a value for the argument when calling the procedure and that the procedure passes a value back to its calling environment after execution.

If you omit IN, OUT, and IN OUT, the argument defaults to IN.

NOCOPY Specify NOCOPY to instruct Oracle to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an index-by table, or a varray to an OUT or IN OUT parameter. (IN parameter values are always passed NOCOPY.)

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter (or assignments made to this parameter may show immediately in a package variable) if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the procedure is exited with an unhandled exception, any assignment made to this parameter may be visible in the caller's variable.

These effects may or may not occur on any particular call. You should use NOCOPY only when these effects would not matter.

datatype Specify the datatype of the argument. An argument can have any datatype supported by PL/SQL.

Datatypes cannot specify length, precision, or scale. For example, VARCHAR2(10) is not valid, but VARCHAR2 is valid. Oracle derives the length, precision, and scale of an argument from the environment from which the procedure is called.

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the procedure executes with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the procedure.

AUTHID CURRENT_USER

Specify CURRENT_USER to indicate that the procedure executes with the privileges of CURRENT_USER. This clause creates an **invoker-rights procedure**.

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the procedure resides.

AUTHID DEFINER

Specify DEFINER to indicate that the procedure executes with the privileges of the owner of the schema in which the procedure resides, and that external names resolve in the schema where the procedure resides. This is the default and creates a **definer-rights procedure**.

See Also:

- *PL/SQL User's Guide and Reference*
- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined

IS | AS Clause

pl/sql_subprogram_body

Declare the procedure in a PL/SQL subprogram body.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for more information on PL/SQL subprograms

call_spec

Use the *call_spec* to map a Java or C method name, parameter types, and return type to their SQL counterparts.

In *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle9i Java Stored Procedures Developer's Guide* for an explanation of the parameters and semantics of the *Java_declaration*
- *Oracle9i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

AS EXTERNAL The AS EXTERNAL clause is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

Examples

Creating a Procedure: Example The following statement creates the procedure remove_emp in the schema hr:

```
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
   tot_emps NUMBER;
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
   tot_emps := tot_emps - 1;
   END;
/
```

The remove_emp procedure removes a specified employee. When you call the procedure, you must specify the employee_id of the employee to be removed. The argument's datatype is NUMBER.

The procedure uses a DELETE statement to remove from the employees table the row of employee_id.

See Also: "Creating a Package Body: Example" on page 14-57 to see how to incorporate this procedure into a package

In the following example, external procedure `c_find_root` expects a pointer as a parameter. Procedure `find_root` passes the parameter by reference using the `BY REFERENCE` phrase:

```
CREATE PROCEDURE find_root
   ( x IN REAL )
   IS LANGUAGE C
      NAME "c_find_root"
      LIBRARY c_utils
      PARAMETERS ( x BY REFERENCE );
```

CREATE PROFILE

Purpose

Use the CREATE PROFILE statement to create a **profile**, which is a set of limits on database resources. If you assign the profile to a user, that user cannot exceed these limits.

> **See Also:** *Oracle9i Database Administrator's Guide* for a detailed description and explanation of how to use password management and protection

Prerequisites

To create a profile, you must have CREATE PROFILE system privilege.

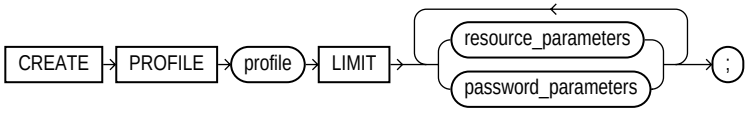
To specify resource limits for a user, you must:

- Enable resource limits dynamically with the ALTER SYSTEM statement or with the initialization parameter RESOURCE_LIMIT. (This parameter does not apply to password resources. Password resources are always enabled.)
- Create a profile that defines the limits using the CREATE PROFILE statement
- Assign the profile to the user using the CREATE USER or ALTER USER statement

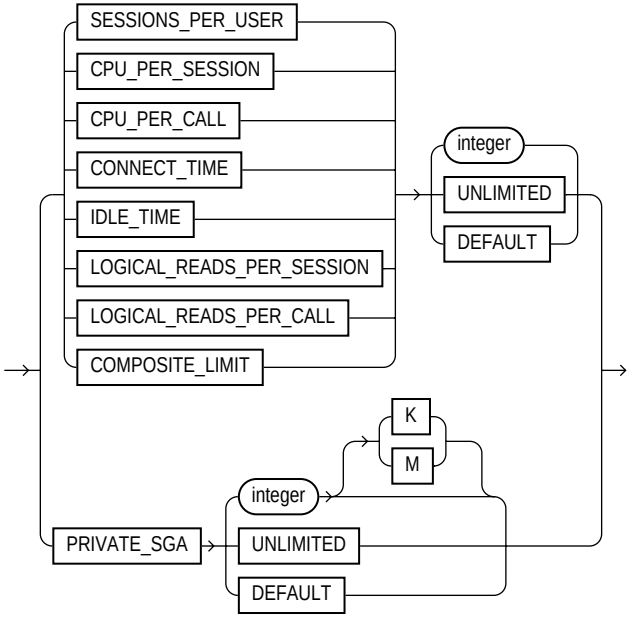
> **See Also:**
>
> - ALTER SYSTEM on page 10-22 for information on enabling resource limits dynamically
> - *Oracle9i Database Reference* for information on the RESOURCE_LIMIT parameter
> - CREATE USER on page 16-32 and ALTER USER on page 12-21 for information on profiles

Syntax

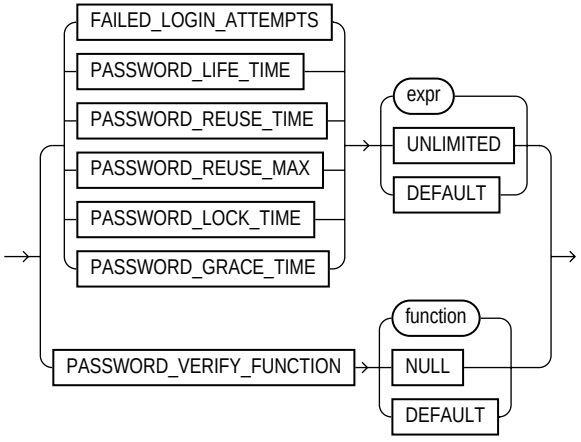
create_profile::=



resource_parameters::=



password_parameters::=



Keywords and Parameters

profile

Specify the name of the profile to be created. Use profiles to limit the database resources available to a user for a single call or a single session.

Oracle enforces resource limits in the following ways:

- If a user exceeds the `CONNECT_TIME` or `IDLE_TIME` session resource limit, Oracle rolls back the current transaction and ends the session. When the user process next issues a call, Oracle returns an error.
- If a user attempts to perform an operation that exceeds the limit for other session resources, Oracle aborts the operation, rolls back the current statement, and immediately returns an error. The user can then commit or roll back the current transaction, and must then end the session.
- If a user attempts to perform an operation that exceeds the limit for a single call, Oracle aborts the operation, rolls back the current statement, and returns an error, leaving the current transaction intact.

Notes:

- You can use fractions of days for all parameters that limit time, with days as units. For example, 1 hour is 1/24 and 1 minute is 1/1440.
- You can specify resource limits for users regardless of whether the resource limits are enabled. However, Oracle does not enforce the limits until you enable them.

See Also: "Creating a Profile: Example" on page 14-74

UNLIMITED

When specified with a resource parameter, `UNLIMITED` indicates that a user assigned this profile can use an unlimited amount of this resource. When specified with a password parameter, `UNLIMITED` indicates that no limit has been set for the parameter.

DEFAULT

Specify DEFAULT if you want to omit a limit for this resource in this profile. A user assigned this profile is subject to the limit for this resource specified in the DEFAULT profile. The DEFAULT profile initially defines unlimited resources. You can change those limits with the ALTER PROFILE statement.

Any user who is not explicitly assigned a profile is subject to the limits defined in the DEFAULT profile. Also, if the profile that is explicitly assigned to a user omits limits for some resources or specifies DEFAULT for some limits, the user is subject to the limits on those resources defined by the DEFAULT profile.

resource_parameters

SESSIONS_PER_USER Specify the number of concurrent sessions to which you want to limit the user.

CPU_PER_SESSION Specify the CPU time limit for a session, expressed in hundredth of seconds.

CPU_PER_CALL Specify the CPU time limit for a call (a parse, execute, or fetch), expressed in hundredths of seconds.

CONNECT_TIME Specify the total elapsed time limit for a session, expressed in minutes.

IDLE_TIME Specify the permitted periods of continuous inactive time during a session, expressed in minutes. Long-running queries and other operations are not subject to this limit.

LOGICAL_READS_PER_SESSION Specify the permitted number of data blocks read in a session, including blocks read from memory and disk.

LOGICAL_READS_PER_CALL Specify the permitted the number of data blocks read for a call to process a SQL statement (a parse, execute, or fetch).

PRIVATE_SGA Specify the amount of private space a session can allocate in the shared pool of the system global area (SGA), expressed in bytes. Use K or M to specify this limit in kilobytes or megabytes.

Note: This limit applies only if you are using Shared Server architecture. The private space for a session in the SGA includes private SQL and PL/SQL areas, but not shared SQL and PL/SQL areas.

COMPOSITE_LIMIT Specify the total resource cost for a session, expressed in **service units**. Oracle calculates the total service units as a weighted sum of CPU_PER_SESSION, CONNECT_TIME, LOGICAL_READS_PER_SESSION, and PRIVATE_SGA.

> **See Also:** ALTER RESOURCE COST on page 9-133 for information on how to specify the weight for each session resource

If you specify *expr* for any of these parameters, the expression can be of any form except scalar subquery expression.

> **See Also:** "Setting Profile Resource Limits: Example" on page 14-75

password_parameters

FAILED_LOGIN_ATTEMPTS Specify the number of failed attempts to log in to the user account before the account is locked.

PASSWORD_LIFE_TIME Specify the number of days the same password can be used for authentication. The password expires if it is not changed within this period, and further connections are rejected.

PASSWORD_REUSE_TIME Specify the number of days before which a password cannot be reused. If you set PASSWORD_REUSE_TIME to an integer value, then you must set PASSWORD_REUSE_MAX to UNLIMITED.

PASSWORD_REUSE_MAX Specify the number of password changes required before the current password can be reused. If you set PASSWORD_REUSE_MAX to an integer value, then you must set PASSWORD_REUSE_TIME to UNLIMITED.

PASSWORD_LOCK_TIME Specify the number of days an account will be locked after the specified number of consecutive failed login attempts.

PASSWORD_GRACE_TIME Specify the number of days after the grace period begins during which a warning is issued and login is allowed. If the password is not changed during the grace period, the password expires.

PASSWORD_VERIFY_FUNCTION The PASSWORD_VERIFY_FUNCTION clause lets a PL/SQL password complexity verification script be passed as an argument to the CREATE PROFILE statement. Oracle provides a default script, but you can create your own routine or use third-party software instead.

- For *function*, specify the name of the password complexity verification routine.
- Specify NULL to indicate that no password verification is performed.

Restrictions on password parameters:

- If PASSWORD_REUSE_TIME is set to an integer value, PASSWORD_REUSE_MAX must be set to UNLIMITED. If PASSWORD_REUSE_MAX is set to an integer value, PASSWORD_REUSE_TIME must be set to UNLIMITED.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to UNLIMITED, then Oracle uses neither of these password resources.
- If PASSWORD_REUSE_MAX is set to DEFAULT and PASSWORD_REUSE_TIME is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_MAX value defined in the DEFAULT profile.
- If PASSWORD_REUSE_TIME is set to DEFAULT and PASSWORD_REUSE_MAX is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_TIME value defined in the DEFAULT profile.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to DEFAULT, then Oracle uses whichever value is defined in the DEFAULT profile.

> **See Also:** "Setting Profile Password Limits: Example" on page 14-75

Examples

Creating a Profile: Example The following statement creates the profile new_profile:

```
CREATE PROFILE new_profile
  LIMIT PASSWORD_REUSE_MAX DEFAULT
        PASSWORD_REUSE_TIME UNLIMITED;
```

Setting Profile Resource Limits: Example The following statement creates the profile app_user:

```
CREATE PROFILE app_user LIMIT
   SESSIONS_PER_USER          UNLIMITED
   CPU_PER_SESSION            UNLIMITED
   CPU_PER_CALL               3000
   CONNECT_TIME               45
   LOGICAL_READS_PER_SESSION  DEFAULT
   LOGICAL_READS_PER_CALL     1000
   PRIVATE_SGA                15K
   COMPOSITE_LIMIT            5000000;
```

If you then assign the app_user profile to a user, the user is subject to the following limits in subsequent sessions:

- The user can have any number of concurrent sessions.
- In a single session, the user can consume an unlimited amount of CPU time.
- A single call made by the user cannot consume more than 30 seconds of CPU time.
- A single session cannot last for more than 45 minutes.
- In a single session, the number of data blocks read from memory and disk is subject to the limit specified in the DEFAULT profile.
- A single call made by the user cannot read more than 1000 data blocks from memory and disk.
- A single session cannot allocate more than 15 kilobytes of memory in the SGA.
- In a single session, the total resource cost cannot exceed 5 million service units. The formula for calculating the total resource cost is specified by the ALTER RESOURCE COST statement.
- Since the system_manager profile omits a limit for IDLE_TIME and for password limits, the user is subject to the limits on these resources specified in the DEFAULT profile.

Setting Profile Password Limits: Example The following statement creates the same app_user2 profile with password limits values set:

```
CREATE PROFILE app_user2 LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LIFE_TIME 60
   PASSWORD_REUSE_TIME 60
```

```
   PASSWORD_REUSE_MAX UNLIMITED
   PASSWORD_VERIFY_FUNCTION verify_function
   PASSWORD_LOCK_TIME 1/24
   PASSWORD_GRACE_TIME 10;
```

This example uses Oracle's password verification function, `verify_function`. Please refer to *Oracle9i Database Administrator's Guide* for information on using this verification function provided by Oracle or designing your own verification function.

CREATE ROLE

Purpose

Use the CREATE ROLE statement to create a **role**, which is a set of privileges that can be granted to users or to other roles. You can use roles to administer database privileges. You can add privileges to a role and then grant the role to a user. The user can then enable the role and exercise the privileges granted by the role.

A role contains all privileges granted to the role and all privileges of other roles granted to it. A new role is initially empty. You add privileges to a role with the GRANT statement.

When you create a role that is NOT IDENTIFIED or is IDENTIFIED EXTERNALLY or BY *password*, Oracle grants you the role with ADMIN OPTION. However, when you create a role IDENTIFIED GLOBALLY, Oracle does not grant you the role.

See Also:

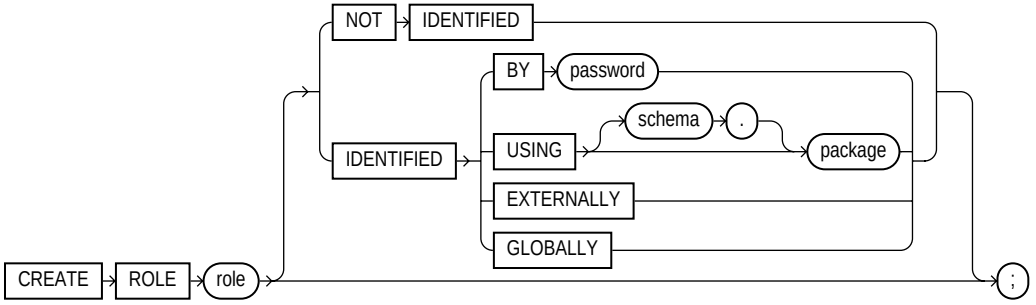
- GRANT on page 17-29 for information on granting roles
- ALTER USER on page 12-21 for information on enabling roles
- ALTER ROLE on page 9-136 for information on modifying a role
- DROP ROLE on page 16-96 for information on removing a role from the database
- SET ROLE on page 18-47 for information on enabling and disabling roles for the current session
- *Oracle9i Heterogeneous Connectivity Administrator's Guide* for a detailed description and explanation of using global roles

Prerequisites

You must have CREATE ROLE system privilege.

Syntax

create_role::=



Keywords and Parameters

role

Specify the name of the role to be created. Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

Some roles are defined by SQL scripts provided on your distribution media.

> **See Also:** GRANT on page 17-29 for a list of these predefined roles

NOT IDENTIFIED Clause

Specify `NOT IDENTIFIED` to indicate that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED Clause

Use the `IDENTIFIED` clause to indicate that a user must be authorized by the specified method before the role is enabled with the `SET ROLE` statement.

BY *password* The `BY` *`password`* clause lets you create a **local role** and indicates that the user must specify the password to Oracle when enabling the role. The password can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

USING *package* The `USING` *`package`* clause lets you create an **application role**, which is a role that can be enabled only by applications using an authorized

package. If you do not specify *schema*, Oracle assumes the package is in your own schema.

EXTERNALLY Specify EXTERNALLY to create an **external role.** An external user must be authorized by an external service (such as an operating system or third-party service) before enabling the role.

Depending on the operating system, the user may have to specify a password to the operating system before the role is enabled.

GLOBALLY Specify GLOBALLY to create a **global role.** A global user must be authorized to use the role by the enterprise directory service before the role is enabled with the SET ROLE statement, or at login.

If you omit both the NOT IDENTIFIED clause and the IDENTIFIED clause, the role defaults to NOT IDENTIFIED.

Examples

Creating a Role: Example The following statement creates the role dw_manager:

```
CREATE ROLE dw_manager;
```

Users who are subsequently granted the dw_manager will inherit all of the privileges that have been granted to this role.

You can add a layer of security to roles by specifying a password, as in the following example:

```
CREATE ROLE dw_manager
   IDENTIFIED BY warehouse;
```

Users who are subsequently granted the dw_manager role must specify the password warehouse to enable the role with the SET ROLE statement.

The following statement creates global role warehouse_user:

```
CREATE ROLE warehouse_user IDENTIFIED GLOBALLY;
```

The following statement creates the same role as an external role:

```
CREATE ROLE warehouse_user IDENTIFIED EXTERNALLY;
```

CREATE ROLLBACK SEGMENT

Purpose

Use the CREATE ROLLBACK SEGMENT statement to create a **rollback segment**, which is an object that Oracle uses to store data necessary to reverse, or undo, changes made by transactions.

The information in this section assumes that your database is running in rollback undo mode (the UNDO_MANAGEMENT initialization parameter is set to MANUAL or not set at all).

If your database is running in Automatic Undo Management mode (the UNDO_MANAGEMENT initialization parameter is set to AUTO), then user-created rollback segments are irrelevant. In this case, Oracle returns an error in response to any CREATE ROLLBACK SEGMENT or ALTER ROLLBACK SEGMENT statement. To suppress these errors, set the UNDO_SUPPRESS_ERRORS parameter to TRUE.

Further, if your database has a locally managed SYSTEM tablespace, then you cannot create rollback segments in any dictionary-managed tablespace. Instead, you must

- Use the Automatic Undo Management feature, which uses undo tablespaces instead of rollback segments to hold undo data, or
- Create locally managed tablespaces to hold the rollback segments.

Oracle Corporation recommends that you use Automatic Undo Management.

Notes:

- A tablespace can have multiple rollback segments. Generally, multiple rollback segments improve performance.
- The tablespace must be online for you to add a rollback segment to it.
- When you create a rollback segment, it is initially offline. To make it available for transactions by your Oracle instance, bring it online using the ALTER ROLLBACK SEGMENT statement. To bring it online automatically whenever you start up the database, add the segment's name to the value of the ROLLBACK_SEGMENTS initialization parameter.

To use objects in a tablespace other than the SYSTEM tablespace:

- If you are running the database in rollback undo mode, at least one rollback segment (other than the SYSTEM rollback segment) must be online.
- If you are running the database in Automatic Undo Management mode, at least one UNDO tablespace must be online.

See Also:

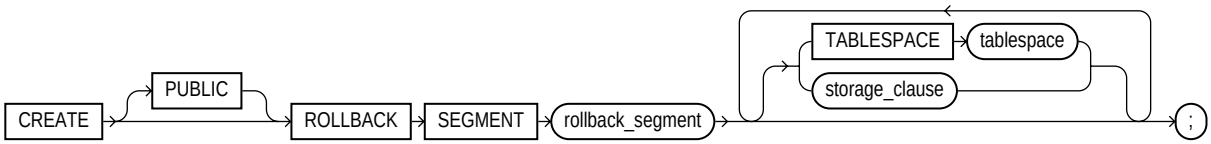
- ALTER ROLLBACK SEGMENT on page 9-138 for information on altering a rollback segment
- DROP ROLLBACK SEGMENT on page 16-97 for information on removing a rollback segment
- *Oracle9i Database Reference* for information on the UNDO_MANAGEMENT and UNDO_SUPPRESS_ERRORS parameters
- *Oracle9i Database Administrator's Guide* for information on Automatic Undo Management mode

Prerequisites

To create a rollback segment, you must have CREATE ROLLBACK SEGMENT system privilege.

Syntax

create_rollback_segment::=



(*storage_clause::=* on page 7-58)

Keyword and Parameters

PUBLIC

Specify PUBLIC to indicate that the rollback segment is public and is available to any instance. If you omit this clause, the rollback segment is private and is available only to the instance naming it in its initialization parameter ROLLBACK_SEGMENTS.

rollback_segment

Specify the name of the rollback segment to be created.

TABLESPACE

Use the TABLESPACE clause to identify the tablespace in which the rollback segment is created. If you omit this clause, Oracle creates the rollback segment in the SYSTEM tablespace.

Note: Oracle must access rollback segments frequently. Therefore, Oracle Corporation strongly recommends that you do not create rollback segments in the SYSTEM tablespace, either explicitly or implicitly (by omitting this clause). In addition, to avoid high contention for the tablespace containing the rollback segment, it should not contain other objects such as tables and indexes, and it should require minimal extent allocation and deallocation.

To achieve these goals, create rollback segments in locally managed tablespaces with autoallocation disabled—that is, in tablespaces created with the EXTENT MANAGEMENT LOCAL clause with the UNIFORM setting. (The AUTOALLOCATE setting is not supported.)

See Also:

- CREATE TABLESPACE on page 15-80
- *Oracle9i Database Administrator's Guide* for more information on creating rollback segments and making them available

storage_clause

The *storage_clause* lets you specify storage characteristics for the rollback segment.

Notes:

- The OPTIMAL parameter of the *storage_clause* is of particular interest, because it applies only to rollback segments.
- You cannot specify the PCTINCREASE parameter of the *storage_clause* with CREATE ROLLBACK SEGMENT.

See Also: *storage_clause* on page 7-56

Examples

Creating a Rollback Segment: Example The following statement creates a rollback segment with default storage values in an appropriately configured tablespace:

```
CREATE TABLESPACE rbs_ts
   DATAFILE 'rbs01.dbf' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 100K;

/* This example and the next will fail if your database is in
   Automatic Undo Mode.
*/
CREATE ROLLBACK SEGMENT rbs_one
   TABLESPACE rbs_ts;
```

The preceding statement is equivalent to the following:

```
CREATE ROLLBACK SEGMENT rbs_one
   TABLESPACE rbs_ts
   STORAGE
   ( INITIAL 10K
     NEXT 10K
     MAXEXTENTS UNLIMITED);
```

CREATE SCHEMA

Purpose

Use the CREATE SCHEMA to create multiple tables and views and perform multiple grants in a single transaction.

To execute a CREATE SCHEMA statement, Oracle executes each included statement. If all statements execute successfully, Oracle commits the transaction. If any statement results in an error, Oracle rolls back all the statements.

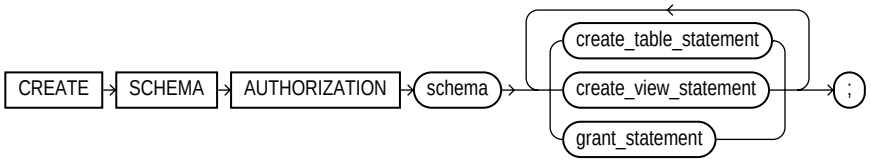
Note: This statement does not actually create a schema. Oracle automatically creates a schema when you create a user (see CREATE USER on page 16-32). This statement lets you populate your schema with tables and views and grant privileges on those objects without having to issue multiple SQL statements in multiple transactions.

Prerequisites

The CREATE SCHEMA statement can include CREATE TABLE, CREATE VIEW, and GRANT statements. To issue a CREATE SCHEMA statement, you must have the privileges necessary to issue the included statements.

Syntax

create_schema::=



Keyword and Parameters

schema

Specify the name of the schema. The schema name must be the same as your Oracle username.

create_table_statement

Specify a CREATE TABLE statement to be issued as part of this CREATE SCHEMA statement. Do not end this statement with a semicolon (or other terminator character).

See Also: CREATE TABLE on page 15-7

create_view_statement

Specify a CREATE VIEW statement to be issued as part of this CREATE SCHEMA statement. Do not end this statement with a semicolon (or other terminator character).

See Also: CREATE VIEW on page 16-39

grant_statement

Specify a GRANT *object_privileges* statement to be issued as part of this CREATE SCHEMA statement. Do not end this statement with a semicolon (or other terminator character).

See Also: GRANT on page 17-29

The CREATE SCHEMA statement supports the syntax of these statements only as defined by standard SQL, rather than the complete syntax supported by Oracle.

The order in which you list the CREATE TABLE, CREATE VIEW, and GRANT statements is unimportant. The statements within a CREATE SCHEMA statement can reference existing objects or objects you create in other statements within the same CREATE SCHEMA statement.

Restriction on *grant_statement*: The syntax of the *parallel_clause* is allowed for a CREATE TABLE statement in CREATE SCHEMA, but parallelism is **not** used when creating the objects.

See Also: the *parallel_clause* of CREATE TABLE on page 15-56

Example

Creating a Schema: Example The following statement creates a schema named oe for the sample order-entry user oe, creates the table new_product, creates the

view new_product_view, and grants SELECT privilege on new_product_view to the sample human resources user hr.

```
CREATE SCHEMA AUTHORIZATION oe
   CREATE TABLE new_product
      (color VARCHAR2(10)  PRIMARY KEY, quantity NUMBER)
   CREATE VIEW new_product_view
      AS SELECT color, quantity FROM new_product WHERE color = 'RED'
   GRANT select ON new_product_view TO hr;
```

CREATE SEQUENCE

Purpose

Use the CREATE SEQUENCE statement to create a **sequence**, which is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, the sequence numbers each user acquires may have gaps because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. Once a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

Once a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn (which returns the current value of the sequence) or the NEXTVAL pseudocolumn (which increments the sequence and returns the new value).

See Also:

- "Pseudocolumns" on page 2-83 for more information on using the CURRVAL and NEXTVAL
- "How to Use Sequence Values" on page 2-84 for information on using sequences
- ALTER SEQUENCE on page 9-142 or DROP SEQUENCE on page 17-2 for information on modifying or dropping a sequence

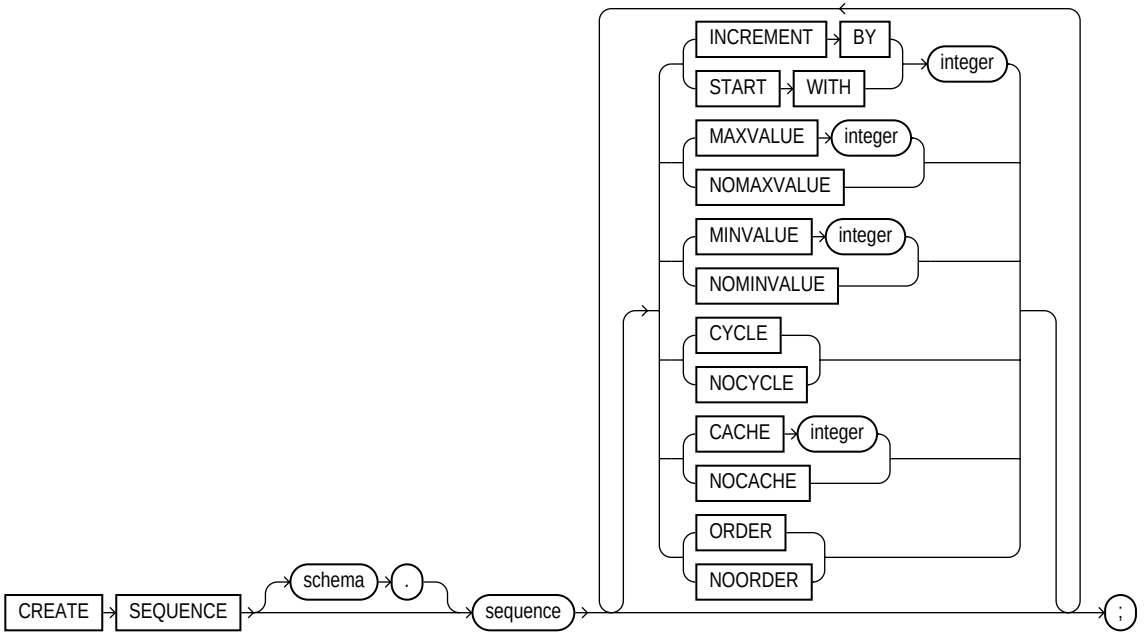
Prerequisites

To create a sequence in your own schema, you must have CREATE SEQUENCE privilege.

To create a sequence in another user's schema, you must have CREATE ANY SEQUENCE privilege.

Syntax

create_sequence::=



Keywords and Parameters

schema

Specify the schema to contain the sequence. If you omit *schema*, Oracle creates the sequence in your own schema.

sequence

Specify the name of the sequence to be created.

If you specify none of the following clauses, you create an ascending sequence that starts with 1 and increases by 1 with no upper limit. Specifying only INCREMENT BY -1 creates a descending sequence that starts with -1 and decreases with no lower limit.

- To create a sequence that **increments without bound**, for ascending sequences, omit the MAXVALUE parameter or specify NOMAXVALUE. For descending sequences, omit the MINVALUE parameter or specify the NOMINVALUE.
- To create a sequence that **stops at a predefined limit**, for an ascending sequence, specify a value for the MAXVALUE parameter. For a descending sequence, specify a value for the MINVALUE parameter. Also specify the NOCYCLE. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- To create a sequence that **restarts after reaching a predefined limit**, specify values for both the MAXVALUE and MINVALUE parameters. Also specify the CYCLE. If you do not specify MINVALUE, then it defaults to NOMINVALUE (that is, the value 1).

Sequence Parameters

INCREMENT BY Specify the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the interval defaults to 1.

START WITH Specify the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence. For descending sequences, the default value is the maximum value of the sequence. This integer value can have 28 or fewer digits.

> **Note:** This value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.

MAXVALUE Specify the maximum value the sequence can generate. This integer value can have 28 or fewer digits. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.

NOMAXVALUE Specify NOMAXVALUE to indicate a maximum value of 10^{27} for an ascending sequence or -1 for a descending sequence. This is the default.

MINVALUE Specify the minimum value of the sequence. This integer value can have 28 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

NOMINVALUE Specify NOMINVALUE to indicate a minimum value of 1 for an ascending sequence or -10^{26} for a descending sequence. This is the default.

CYCLE Specify CYCLE to indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum.

NOCYCLE Specify NOCYCLE to indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.

CACHE Specify how many values of the sequence Oracle preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for CACHE must be less than the value determined by the following formula:

```
(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)
```

If a system failure occurs, all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the CACHE parameter.

Note: Oracle Corporation recommends using the CACHE setting to enhance performance if you are using sequences in a Real Application Clusters environment.

NOCACHE Specify NOCACHE to indicate that values of the sequence are not preallocated.

If you omit both CACHE and NOCACHE, Oracle caches 20 sequence numbers by default.

ORDER Specify ORDER to guarantee that sequence numbers are generated in order of request. You may want to use this clause if you are using the sequence numbers

as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys.

ORDER is necessary only to guarantee ordered generation if you are using Oracle with Real Application Clusters. If you are using exclusive mode, sequence numbers are always generated in order.

NOORDER Specify NOORDER if you do not want to guarantee sequence numbers are generated in order of request. This is the default.

Example

Creating a Sequence: Example The following statement creates the sequence customers_seq in the sample schema oe. This sequence could be used to provide customer ID numbers when rows are added to the customers table.

```
CREATE SEQUENCE customers_seq
 START WITH     1000
 INCREMENT BY   1
 NOCACHE
 NOCYCLE;
```

The first reference to customers_seq.nextval returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

CREATE SPFILE

Purpose

Use the CREATE SPFILE statement to create a **server parameter file** from a client-side initialization parameter file. Server parameter files are binary files that exist only on the server and are called from client locations to start up the database.

Server parameter files let you make persistent changes to individual parameters. When you use a server parameter file, you can specify in an ALTER SYSTEM SET *parameter* statement that the new parameter value should be persistent. This means that the new value applies not only in the current instance, but also to any instances that are started up subsequently. Traditional client-side parameter files do not let you make persistent changes to parameter values. Because they are located on the server, these files allow for automatic database tuning by Oracle and for backup by Recovery Manager (RMAN).

To use a server parameter file when starting up the database, you must create it from a traditional text initialization parameter file using the CREATE SPFILE statement.

All instances in an Real Application Clusters environment must use the same server parameter file. However, when otherwise permitted, individual instances can have different settings of the same parameter within this one file. Instance-specific parameter definitions are specified as *SID.parameter = value*, where *SID* is the instance identifier.

The method of starting up the database with a server parameter file depends on whether you create a default or nondefault server parameter file. Please refer to "Creating a Server Parameter File: Examples" on page 14-94 for examples of how to use server parameter files.

See Also:

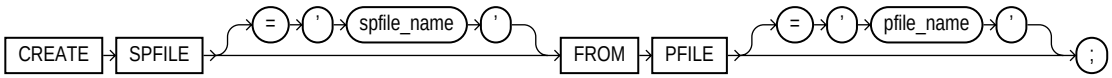
- CREATE PFILE on page 14-60 for information on creating a regular text parameter file from a binary server parameter file
- *Oracle9i Database Administrator's Guide* for information on pre-Oracle9*i* initialization parameter files and Oracle9*i* server parameter files
- *Oracle9i Real Application Clusters Administration* for information on using server parameter files in a Real Application Clusters environment

Prerequisites

You must have the SYSDBA or the SYSOPER system privilege to execute this statement. You can execute this statement before or after instance startup. However, if you have already started an instance using *spfile_name*, you cannot specify the same *spfile_name* in this statement.

Syntax

create_spfile::=



Keywords and Parameters

spfile_name

This clause lets you specify a name for the server parameter file you are creating.

- If you do not specify *spfile_name*, Oracle uses the platform-specific default server parameter filename. If *spfile_name* already exists on the server, this statement will overwrite it. When using a default server parameter file, you start up the database without referring to the file by name.
- If you do specify *spfile_name*, you are creating a nondefault server parameter file. In this case, to start up the database, you must first create a single-line traditional parameter file that points to the server parameter file, and then name the single-line file in your STARTUP command.

See Also:

- "Creating a Server Parameter File: Examples" on page 14-94 for information on starting up the database with default and nondefault server parameter files
- *Oracle9i Database Administrator's Guide for Windows* (or other appropriate operating system specific documentation) for default parameter file names

pfile_name

Specify the traditional initialization parameter file from which you want to create a server parameter file.

- If you specify *pfile_name*, the parameter file must reside on the server. If it does not reside in the default directory for parameter files on your operating system, you must specify the full path.
- If you do not specify *pfile_name*, Oracle looks in the default directory for parameter files on your operating system for the default parameter filename, and uses that file. If that file does not exist in the expected directory, Oracle returns an error.

> **Note:** In a Real Application Clusters environment, you must first combine all instance parameter files into one file before specifying it in this statement to create a server parameter file. For information on accomplishing this step, see *Oracle9i Real Application Clusters Setup and Configuration*.

Examples

Creating a Server Parameter File: Examples The following example creates a default server parameter file from a client initialization parameter file named `t_init1.ora`:

```
CREATE SPFILE
   FROM PFILE = '$ORACLE_HOME/work/t_init1.ora';
```

> **Note:** Typically you will need to specify the full path and filename for parameter files on your operating system. Please refer to your Oracle operating system documentation for path information.

When you create a default server parameter file, you subsequently start up the database using that server parameter file by using the SQL*Plus command `STARTUP` without the `PFILE` parameter, as follows:

```
STARTUP
```

The following example creates a nondefault server parameter file `s_params.ora` from a client initialization file named `t_init1.ora`:

```
CREATE SPFILE = 's_params.ora'
   FROM PFILE = '$ORACLE_HOME/work/t_init1.ora';
```

When you create a nondefault server parameter file, you subsequently start up the database by first creating a traditional parameter file containing the following single line:

```
spfile = 's_params.ora'
```

The name of this parameter file must comply with the naming conventions of your operating system. You then use the single-line parameter file in the STARTUP command. The following example shows how to start up the database, assuming that the single-line parameter file is named new_param.ora:

```
STARTUP PFILE=new_param.ora
```

15

SQL Statements: CREATE SYNONYM to CREATE TRIGGER

This chapter contains the following SQL statements:

- CREATE SYNONYM
- CREATE TABLE
- CREATE TABLESPACE
- CREATE TEMPORARY TABLESPACE
- CREATE TRIGGER

CREATE SYNONYM

Purpose

Use the CREATE SYNONYM statement to create a **synonym**, which is an alternative name for a table, view, sequence, procedure, stored function, package, materialized view, Java class schema object, user-defined object type, or another synonym.

Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view. However, synonyms are not a substitute for privileges on database objects. Such privileges must be granted to a user before the user can use the synonym.

You can refer to synonyms in the following DML statements: SELECT, INSERT, UPDATE, DELETE, EXPLAIN PLAN, and LOCK TABLE.

You can refer to synonyms in the following DDL statements: AUDIT, NOAUDIT, GRANT, REVOKE, and COMMENT.

> **See Also:** *Oracle9i Database Concepts* for general information on synonyms

Prerequisites

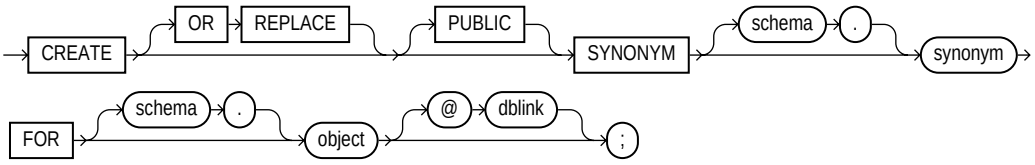
To create a private synonym in your own schema, you must have CREATE SYNONYM system privilege.

To create a private synonym in another user's schema, you must have CREATE ANY SYNONYM system privilege.

To create a PUBLIC synonym, you must have CREATE PUBLIC SYNONYM system privilege.

Syntax

create_synonym::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the synonym if it already exists. Use this clause to change the definition of an existing synonym without first dropping it.

Restriction on replacing a synonym: You cannot use the OR REPLACE clause for a type synonym that has any dependent tables or valid user-defined object types.

PUBLIC

Specify PUBLIC to create a public synonym. Public synonyms are accessible to all users. However each user must have appropriate privileges on the underlying object in order to use the synonym.

Oracle uses a public synonym only when resolving references to an object if the object is not prefaced by a schema and the object is not followed by a database link.

If you omit this clause, then the synonym is private and is accessible only within its schema. A private synonym name must be unique in its schema.

Notes on creating public synonyms:

- If you create a public synonym and it subsequently has dependent tables or dependent valid user-defined object types, then you cannot subsequently create another database object of the same name as the synonym in the same schema as the dependent objects.
- Take care not to create a public synonym with the same name as an existing schema. If you do so, then all PL/SQL units that use that name will be invalidated.

schema

Specify the schema to contain the synonym. If you omit *schema*, then Oracle creates the synonym in your own schema. You cannot specify a schema for the synonym if you have specified PUBLIC.

synonym

Specify the name of the synonym to be created.

Caution: The functional maximum length of the synonym name is 32 bytes. Names longer than 30 bytes are permitted for Java functionality only. If you specify a name longer than 30 bytes, then Oracle encrypts the name and places a representation of the encryption in the data dictionary. The actual encryption is not accessible, and you cannot use either your original specification or the data dictionary representation as the synonym name.

See Also: "CREATE SYNONYM Examples" on page 15-5 and "Resolution of Synonyms Example" on page 15-6

FOR Clause

Specify the object for which the synonym is created. The schema object for which you are creating the synonym can be of the following types:

- Table or object table
- View or object view
- Sequence
- Stored procedure, function, or package
- Materialized view
- Java class schema object
- User-defined object type
- Synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restriction on the FOR clause: The schema object cannot be contained in a package.

schema Specify the schema in which the object resides. If you do not qualify object with *schema*, then Oracle assumes that the schema object is in your own schema.

Note: If you are creating a synonym for a procedure or function on a remote database, then you must specify *schema* in this CREATE statement. Alternatively, you can create a local public synonym on the database where the object resides. However, the database link must then be included in all subsequent calls to the procedure or function.

---

dblink You can specify a complete or partial database link to create a synonym for a schema object on a remote database where the object is located. If you specify *dblink* and omit *schema*, then the synonym refers to an object in the schema specified by the database link. Oracle Corporation recommends that you specify the schema containing the object in the remote database.

If you omit *dblink*, then Oracle assumes the object is located on the local database.

Restriction on database links: You cannot specify *dblink* for a Java class synonym.

See Also:

- "Referring to Objects in Remote Databases" on page 2-118 for more information on referring to database links
- CREATE DATABASE LINK on page 13-35 for more information on creating database links

Examples

CREATE SYNONYM Examples To define the synonym `offices` for the table `locations` in the schema `hr`, issue the following statement:

```
CREATE SYNONYM offices
   FOR hr.locations;
```

To create a `PUBLIC` synonym for the `employees` table in the schema `hr` on the remote `SALES` database, you could issue the following statement:

```
CREATE PUBLIC SYNONYM employees
   FOR hr.employees@sales;
```

A synonym may have the same name as the base table, provided the base table is contained in another schema.

Resolution of Synonyms Example Oracle attempts to resolve references to objects at the schema level before resolving them at the PUBLIC synonym level. For example, the schemas oe and sh both contain tables named customers. In the next example, user SYSTEM creates a PUBLIC synonym named customers for oe.customers:

```
CREATE PUBLIC SYNONYM customers FOR oe.customers;
```

If the user sh then issues the following statement, then Oracle returns the count of rows from sh.customers:

```
SELECT COUNT(*) FROM customers;
```

To retrieve the count of rows from oe.customers, the user sh must preface customers with the schema name. (The user sh must have select permission on oe.customers as well.)

```
SELECT COUNT(*) FROM oe.customers;
```

If the user hr's schema does not contain an object named customers, and if hr has select permission on oe.customers, then hr can access the customers table in oe's schema by using the public synonym customers:

```
SELECT COUNT(*) FROM customers;
```

CREATE TABLE

Purpose

Use the CREATE TABLE statement to create one of the following types of tables:

- A **relational table** is the basic structure to hold user data.
- An **object table** is a table that uses an object type for a column definition. An object table is a table explicitly defined to hold object instances of a particular type.

You can also create an object type and then use it in a column when creating a relational table.

Tables are created with no data unless a query is specified. You can add rows to a table with the INSERT statement. After creating a table, you can define additional columns, partitions, and integrity constraints with the ADD clause of the ALTER TABLE statement. You can change the definition of an existing column or partition with the MODIFY clause of the ALTER TABLE statement.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals*, *Oracle9i Database Administrator's Guide*, and CREATE TYPE on page 16-3 for more information about creating objects
- ALTER TABLE on page 11-2

Prerequisites

To create a **relational table** in your own schema, you must have the CREATE TABLE system privilege. To create a table in another user's schema, you must have CREATE ANY TABLE system privilege. Also, the owner of the schema to contain the table must have either space quota on the tablespace to contain the table or UNLIMITED TABLESPACE system privilege.

In addition to these table privileges, to create an **object table** (or a relational table with an object type column), the owner of the table must have the EXECUTE object privilege in order to access all types referenced by the table, or you must have the EXECUTE ANY TYPE system privilege. These privileges must be granted explicitly and not acquired through a role.

Additionally, if the table owner intends to grant access to the table to other users, then the owner must have been granted the EXECUTE privileges on the referenced

types with the GRANT OPTION, or have the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Without these privileges, the table owner has insufficient privileges to grant access to the table to other users.

To enable a unique or primary key constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle creates an index on the columns of the unique or primary key in the schema containing the table.

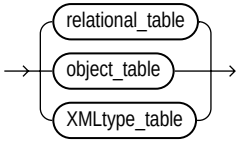
To create an external table, you must have the READ object privilege on the directory in which the external data resides.

See Also:

- CREATE INDEX on page 13-62
- *Oracle9i Application Developer's Guide - Fundamentals* for more information about the privileges required to create tables using types

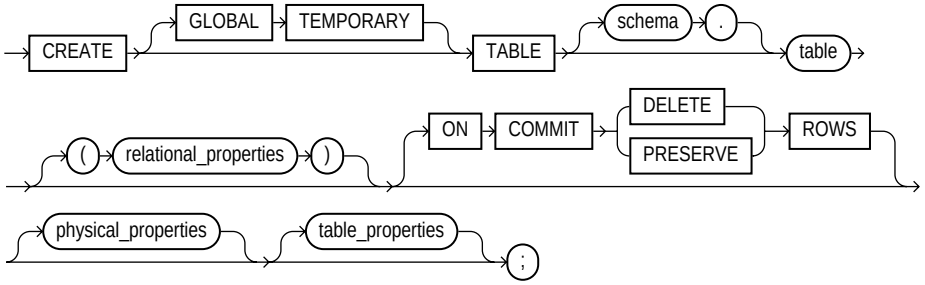
Syntax

create_table::=



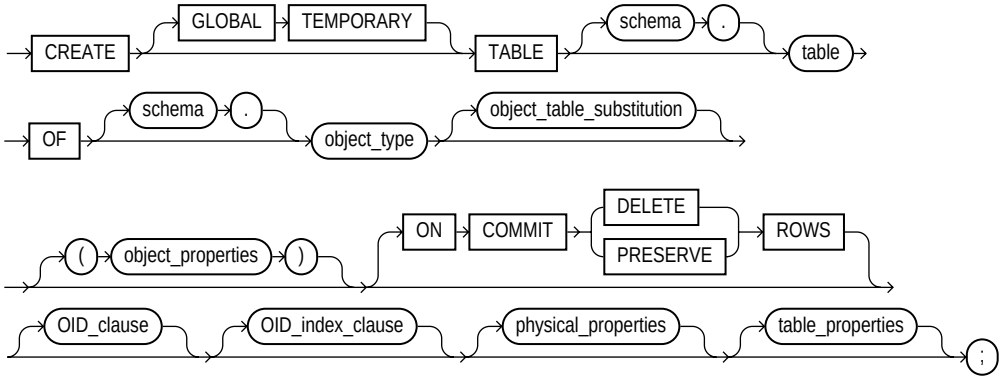
(*relational_table::=* on page 15-8, *object_table::=* on page 15-9, *XMLType_table::=* on page 15-9)

relational_table::=



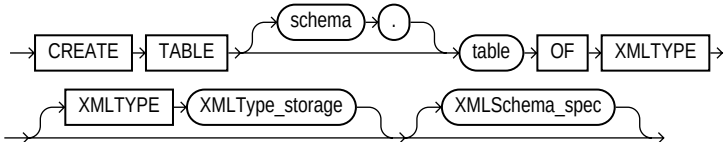
(*relational_properties::=* on page 15-10, *physical_properties::=* on page 15-11, *table_properties::=* on page 15-12)

object_table::=



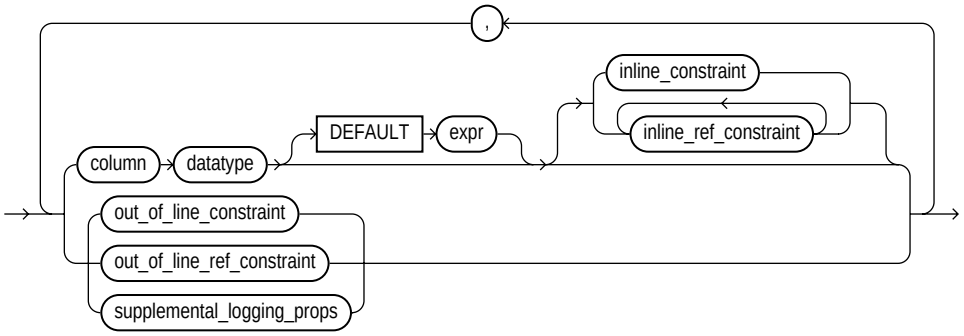
(*object_table_substitution::=* on page 15-10, *object_properties::=* on page 15-10, *OID_clause::=* on page 15-10, *OID_index_clause::=* on page 15-11, *physical_properties::=* on page 15-11, *table_properties::=* on page 15-12)

XMLType_table::=



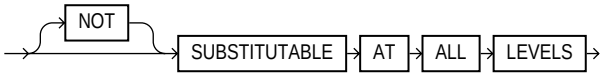
(*XMLType_storage::=* on page 15-16, *XMLSchema_spec::=* on page 15-16)

relational_properties::=

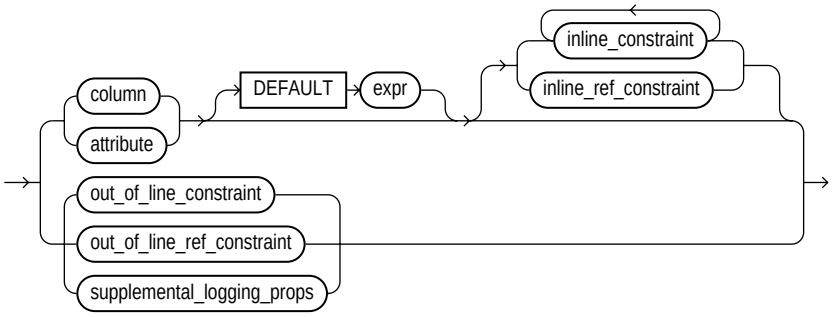


(*constraints::=* on page 7-6)

object_table_substitution::=

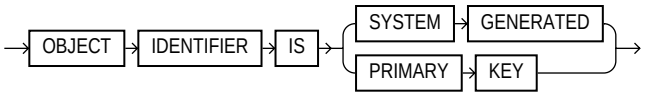


object_properties::=

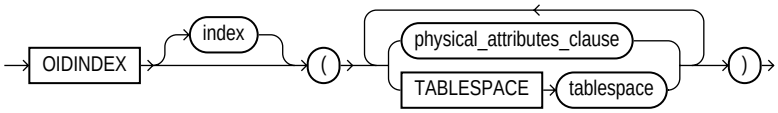


(*constraints::=* on page 7-6, *supplemental_logging_props::=* on page 15-17)

OID_clause::=

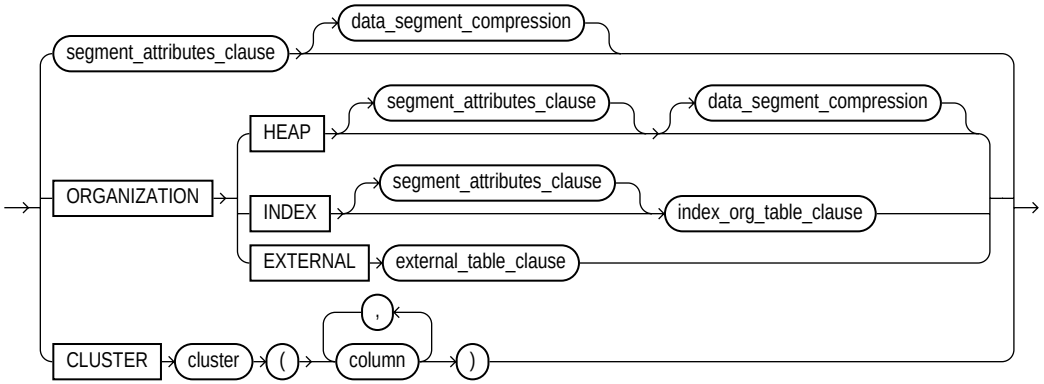


OID_index_clause::=



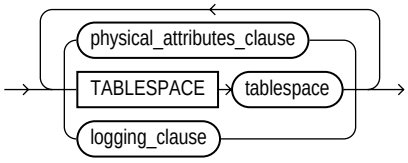
(*physical_attributes_clause::=* on page 15-12)

physical_properties::=



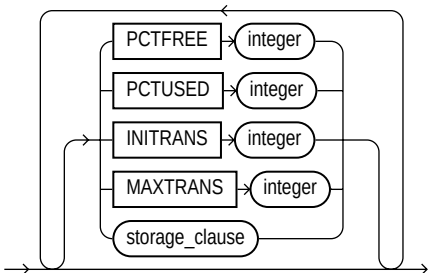
(*segment_attributes_clause::=* on page 15-11, *data_segment_compression::=* on page 15-12, *index_org_table_clause::=* on page 15-16, *external_table_clause::=* on page 15-17)

segment_attributes_clause::=



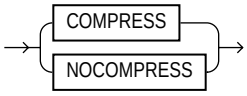
(*physical_attributes_clause::=* on page 15-12, *logging_clause::=* on page 15-15)

physical_attributes_clause::=

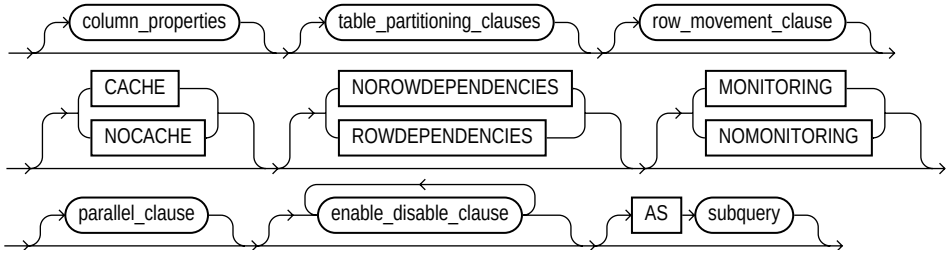


(*storage_clause* on page 7-56)

data_segment_compression::=

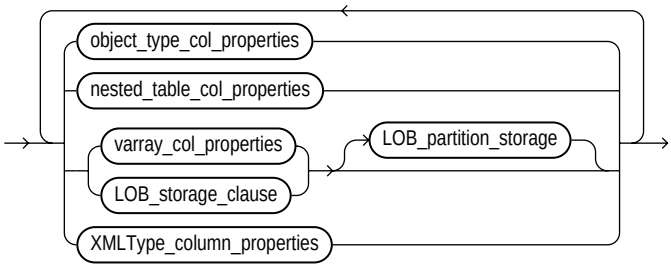


table_properties::=



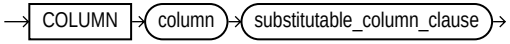
(*table_partitioning_clauses::=* on page 15-18, *parallel_clause::=* on page 15-22, *enable_disable_clause::=* on page 15-23, *subquery::=* on page 18-5)

column_properties::=

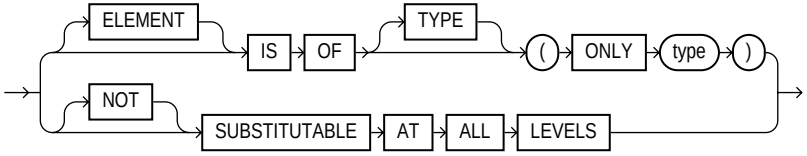


(*object_type_col_properties::=* on page 15-13, *nested_table_col_properties::=* on page 15-13, *varray_col_properties::=* on page 15-14, *LOB_storage_clause::=* on page 15-14, *LOB_partition_storage::=* on page 15-15, *XMLType_column_properties::=* on page 15-16)

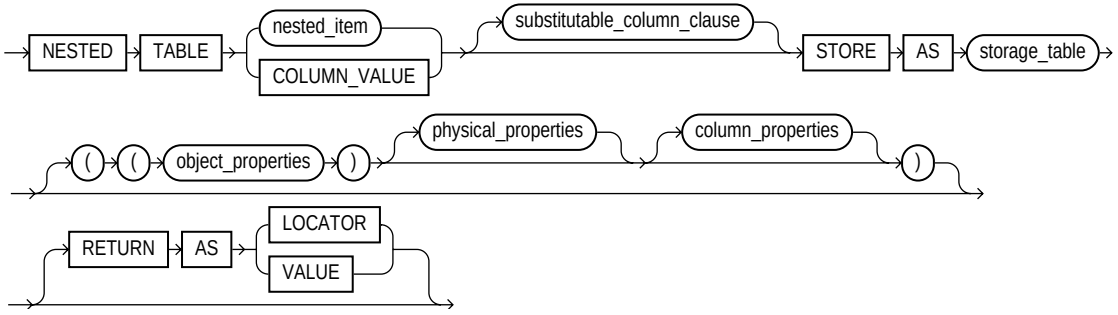
object_type_col_properties::=



substitutable_column_clause::=

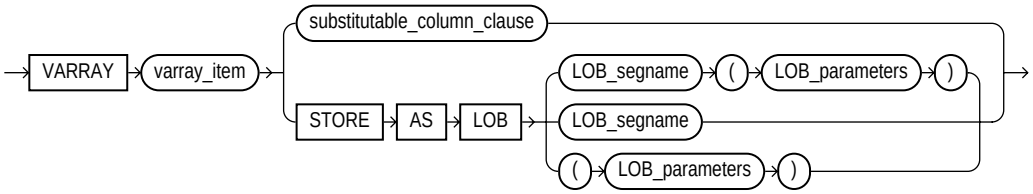


nested_table_col_properties::=



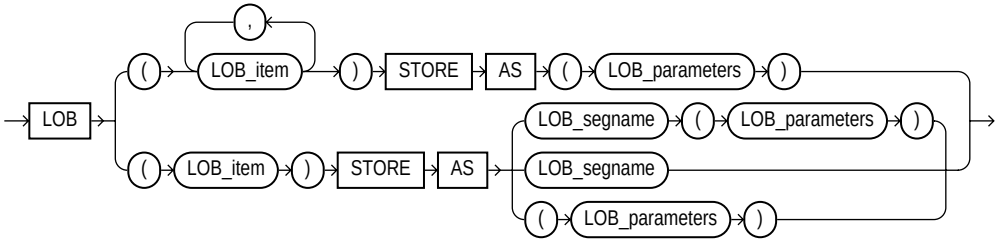
(*substitutable_column_clause::=* on page 15-13, *object_properties::=* on page 15-10, *physical_properties::=* on page 15-11, *column_properties::=* on page 15-13)

varray_col_properties::=



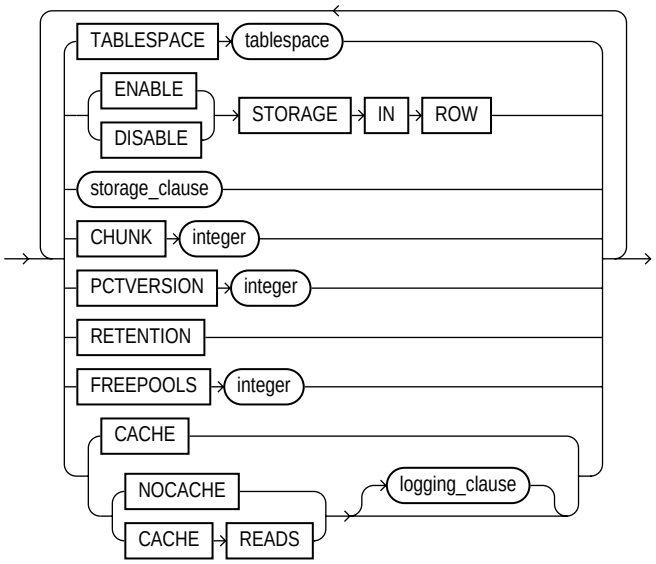
(*substitutable_column_clause::=* on page 15-13, *LOB_parameters::=* on page 15-15)

LOB_storage_clause::=



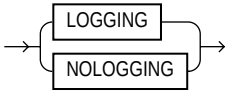
(*LOB_parameters::=* on page 15-15)

LOB_parameters::=

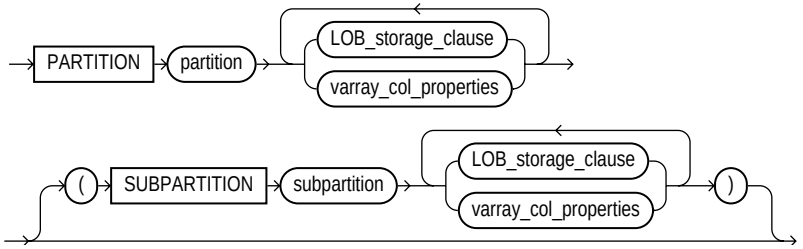


(*storage_clause::=* on page 7-58)

logging_clause::=

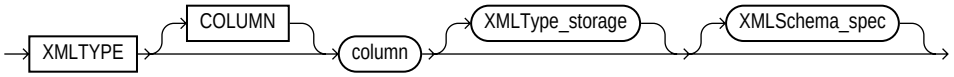


LOB_partition_storage::=



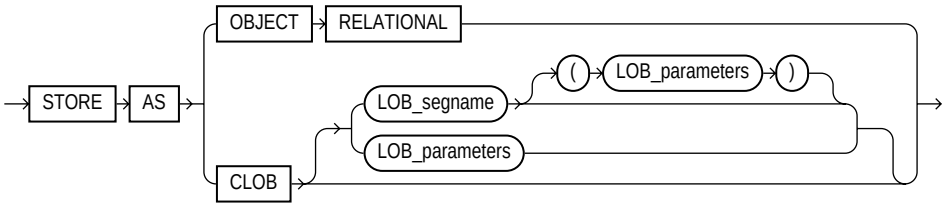
(*LOB_storage_clause::=* on page 15-14, *varray_col_properties::= on page 15-14*)

XMLType_column_properties::=



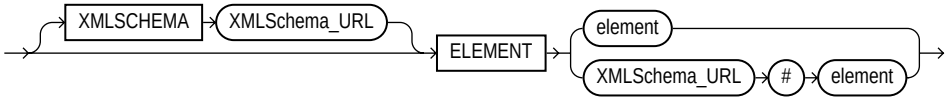
(*XMLType_storage::=* on page 15-16, *XMLSchema_spec::=* on page 15-16)

XMLType_storage::=

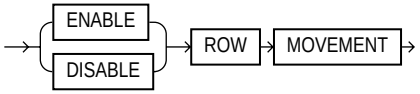


(*LOB_parameters::=* on page 15-15)

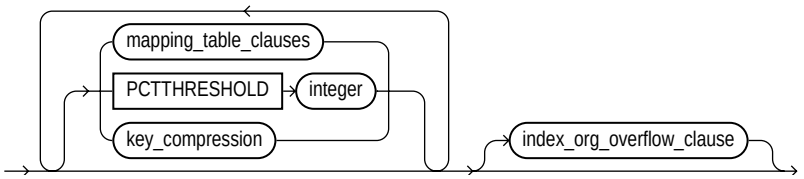
XMLSchema_spec::=



row_movement_clause::=

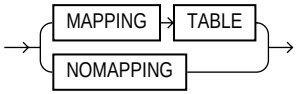


index_org_table_clause::=

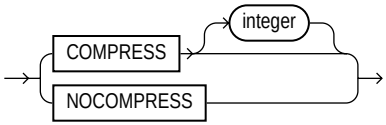


(*mapping_table_clause::=* on page 15-17, *key_compression::=* on page 15-17, *index_org_overflow_clause::=* on page 15-17)

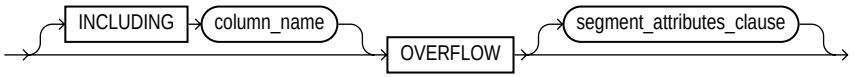
mapping_table_clause::=



key_compression::=

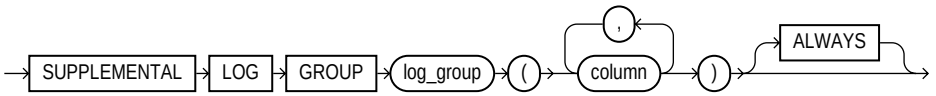


index_org_overflow_clause::=



(*segment_attributes_clause::=* on page 15-11)

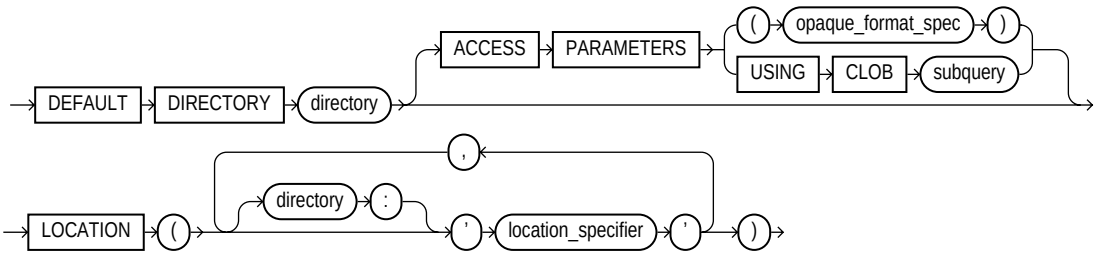
supplemental_logging_props::=



external_table_clause::=

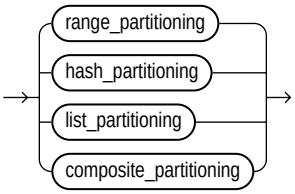
(*external_data_properties::=* on page 15-18)

external_data_properties::=



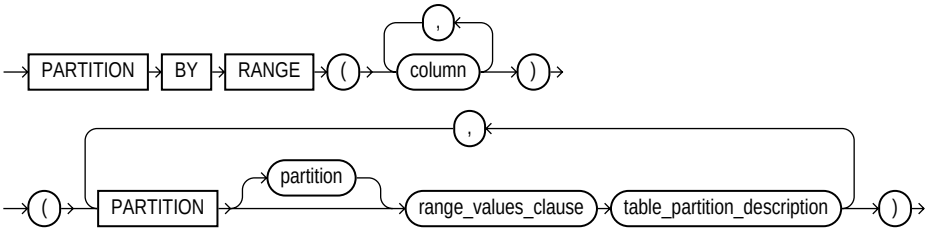
(*opaque_format_spec*: See *Oracle9i Database Utilities* for information on how to specify values for the *opaque_format_spec*.)

table_partitioning_clauses::=



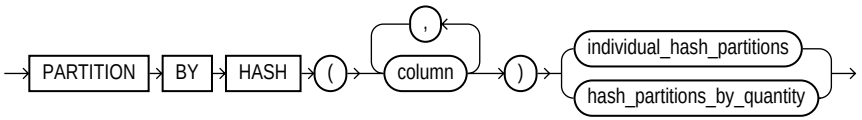
(*range_partitioning::=* on page 15-18, *hash_partitioning::=* on page 15-19, *list_partitioning::=* on page 15-19, *composite_partitioning::=* on page 15-19)

range_partitioning::=



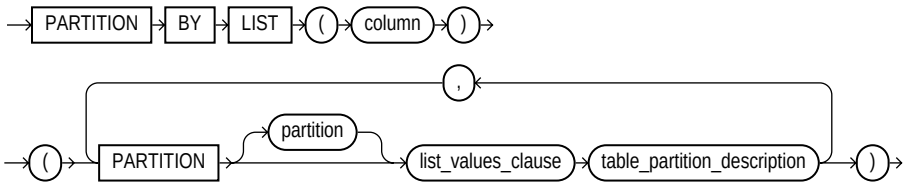
(*range_values_clause::=* on page 15-21, *table_partition_description::= on page 15-21*)

hash_partitioning::=



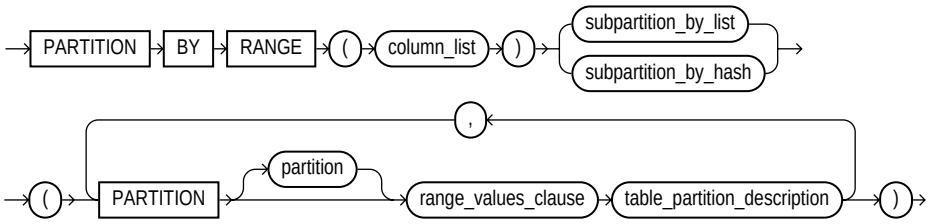
(*individual_hash_partitions::=* on page 15-20, *hash_partitions_by_quantity::=* on page 15-20)

list_partitioning::=



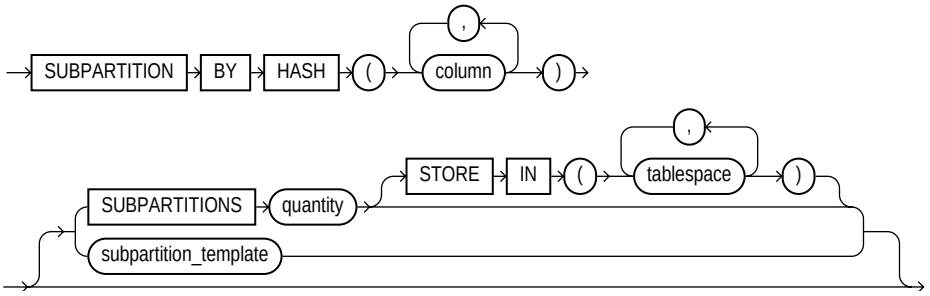
(*list_values_clause::=* on page 15-21, *table_partition_description::= on page 15-21*)

composite_partitioning::=



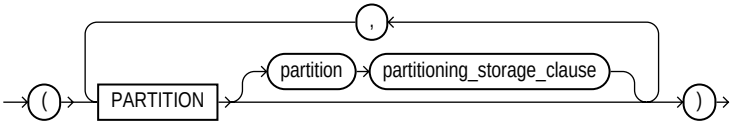
(*subpartition_by_list::=* on page 15-20, *subpartition_by_hash::=* on page 15-20, *range_values_clause::=* on page 15-21, *table_partition_description::= on page 15-21*)

subpartition_by_hash::=



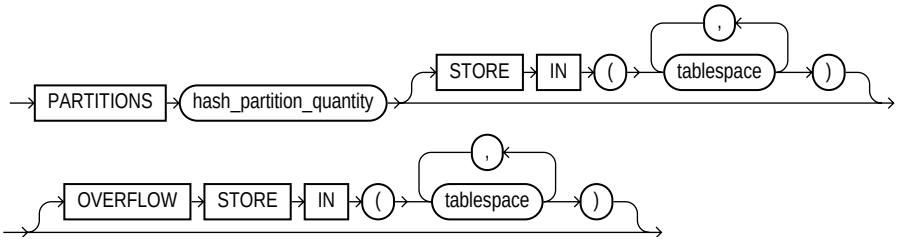
(*subpartition_template::=* on page 15-21)

individual_hash_partitions::=

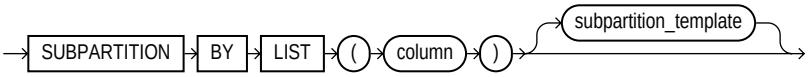


(*partitioning_storage_clause::=* on page 15-22)

hash_partitions_by_quantity::=

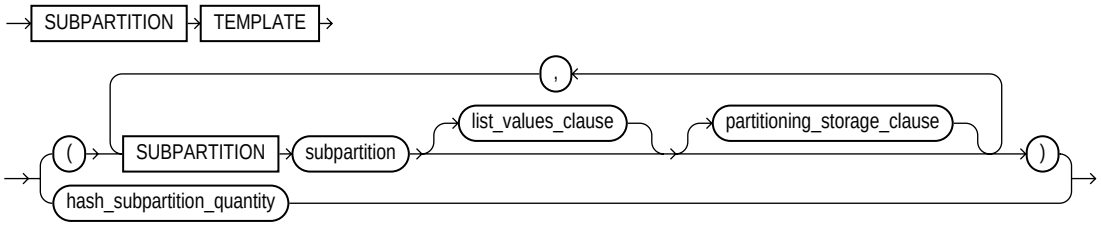


subpartition_by_list::=



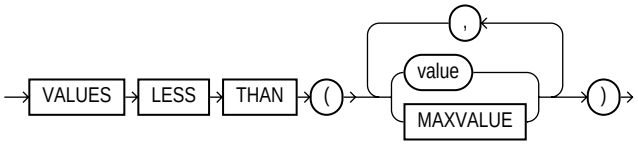
(*subpartition_template::=* on page 15-21)

subpartition_template::=

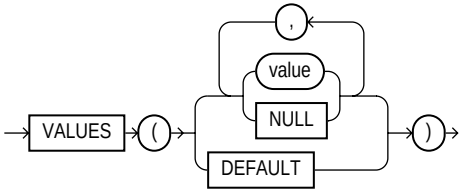


(*list_values_clause::=* on page 15-21, *partitioning_storage_clause::=* on page 15-22)

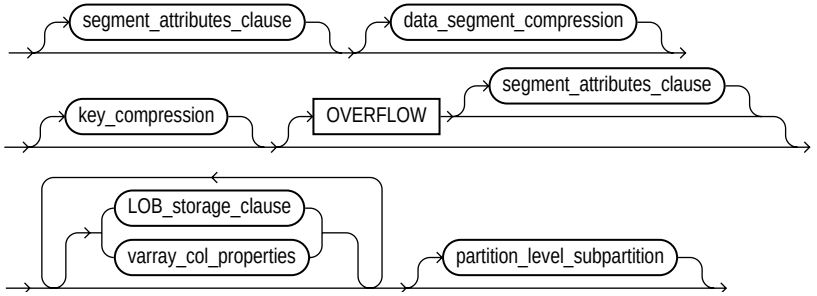
range_values_clause::=



list_values_clause::=

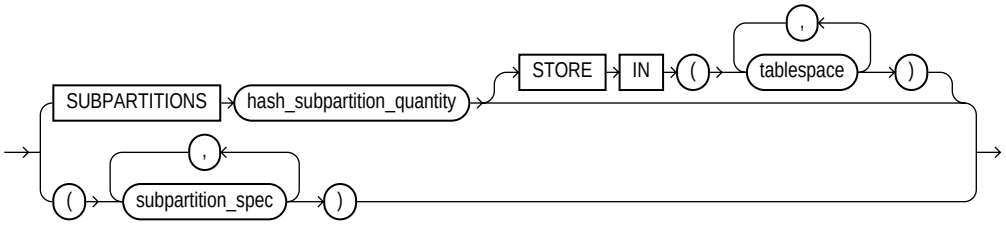


table_partition_description::=



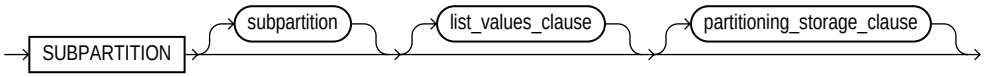
(*segment_attributes_clause::=* on page 15-11, *data_segment_compression::=* on page 15-12, *LOB_storage_clause::=* on page 15-14, *varray_col_properties::=* on page 15-14, *partition_level_subpartition::=* on page 15-22)

partition_level_subpartition::=



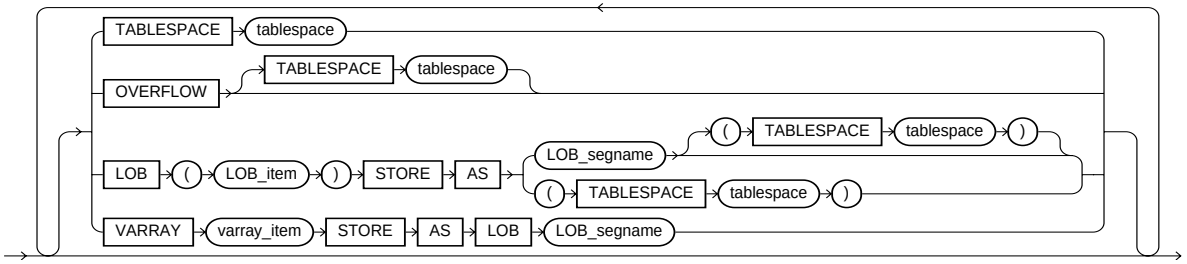
(*subpartition_spec::=* on page 15-22)

subpartition_spec::=

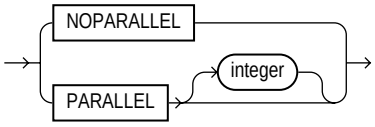


(*list_values_clause::=* on page 15-21, *partitioning_storage_clause::=* on page 15-22)

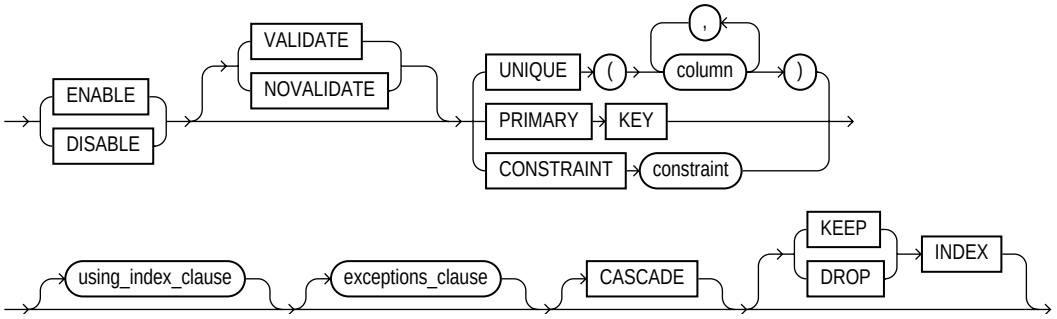
partitioning_storage_clause::=



parallel_clause::=

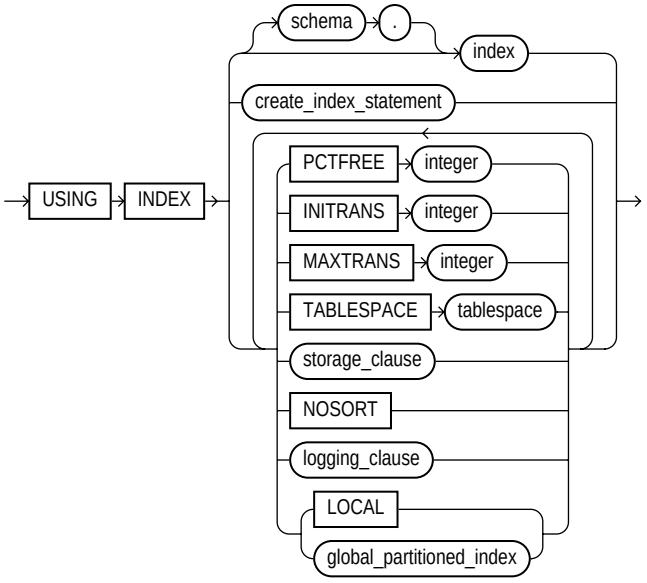


enable_disable_clause::=



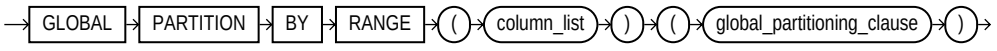
(*using_index_clause::=* on page 15-23, *exceptions_clause* not supported in CREATE TABLE statements)

using_index_clause::=



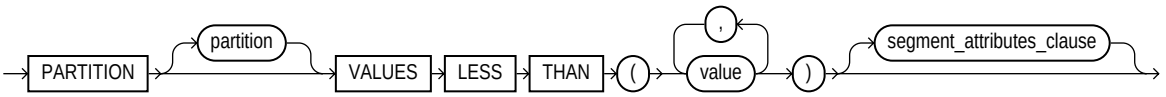
(*create_index::=* on page 13-63, *logging_clause::=* on page 7-46, *global_partitioned_index::=* on page 15-23)

global_partitioned_index::=



(*global_partitioning_clause::=* on page 15-24)

global_partitioning_clause::=



(*segment_attributes_clause::=* on page 15-11)

Keywords and Parameters

relational_table

GLOBAL TEMPORARY

Specify GLOBAL TEMPORARY to indicate that the table is temporary and that its definition is visible to all sessions. The data in a temporary table is visible only to the session that inserts the data into the table.

A temporary table has a definition that persists the same as the definitions of regular tables, but it contains either **session-specific** or **transaction-specific** data. You specify whether the data is session- or transaction-specific with the ON COMMIT keywords.

> **See Also:** *Oracle9i Database Concepts* for information on temporary tables and "Temporary Table Example" on page 15-66

Restrictions on temporary tables:

- Temporary tables cannot be partitioned, clustered, or index organized.
- You cannot specify any foreign key constraints on temporary tables.
- Temporary tables cannot contain columns of nested table or varray type.
- You cannot specify the following clauses of the *LOB_storage_clause*: TABLESPACE, *storage_clause*, *logging_clause*, MONITORING or NOMONITORING, or *LOB_index_clause*.
- Parallel DML and parallel queries are not supported for temporary tables. (Parallel hints are ignored. Specification of the *parallel_clause* returns an error.)

- You cannot specify the *segment_attributes_clause*, *nested_table_col_properties*, or *parallel_clause*.
- Distributed transactions are not supported for temporary tables.

schema

Specify the schema to contain the table. If you omit *schema*, then Oracle creates the table in your own schema.

table

Specify the name of the table (or object table) to be created.

See Also: "General Examples" on page 15-65

relational_properties

The relational properties describe the components of a relational table.

column

Specify the name of a column of the table.

If you also specify AS *subquery*, then you can omit *column* and *datatype* unless you are creating an index-organized table. If you specify AS *subquery* when creating an index-organized table, then you must specify *column*, and you must omit *datatype*.

The absolute maximum number of columns in a table is 1000. However, when you create an object table (or a relational table with columns of object, nested table, varray, or REF type), Oracle maps the columns of the user-defined types to relational columns, creating in effect "hidden columns" that count toward the 1000-column limit.

datatype

Specify the datatype of a column.

Note: You can omit *datatype* under these conditions:

- If you also specify AS *subquery*. (If you are creating an index-organized table and you specify AS *subquery*, you *must* omit the datatype.)
- If the statement also designates the column as part of a foreign key in a referential integrity constraint. (Oracle automatically assigns to the column the datatype of the corresponding column of the referenced key of the referential integrity constraint.)

Restriction on *datatype*: You can specify a column of type ROWID, but Oracle does not guarantee that the values in such columns are valid rowids.

> **See Also:** "Datatypes" on page 2-2 for information on Oracle-supplied datatypes

DEFAULT

The DEFAULT clause lets you specify a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The datatype of the expression must match the datatype of the column. The column must also be long enough to hold this expression.

The DEFAULT expression can include any SQL function as long as the function does not return a literal argument, a column reference, or a nested function invocation.

Restriction on DEFAULT: A DEFAULT expression cannot contain references to PL/SQL functions or to other columns, the pseudocolumns LEVEL, PRIOR, and ROWNUM, or date constants that are not fully specified.

> **See Also:** "About SQL Expressions" on page 4-2 for the syntax of *expr*

inline_ref_constraint and *out_of_line_ref_constraint*

These clauses let you describe a column of type REF. The only difference between these clauses is that you specify *out_of_line_ref_constraint* from the table level, so you must identify the REF column or attribute you are defining. You specify *inline_ref_constraint* after you have already identified the REF column or attribute.

> **See Also:** *constraints* on page 7-5 for syntax and description of these constraints, as well as examples

inline_constraint

Use the *inline_constraint* to define an integrity constraint as part of the column definition.

You can create UNIQUE, PRIMARY KEY, and REFERENCES constraints on scalar attributes of object type columns. You can also create NOT NULL constraints on object type columns, and CHECK constraints that reference object type columns or any attribute of an object type column.

> **See Also:** *constraints* on page 7-5 for syntax and description of these constraints, as well as examples

out_of_line_constraint

Use the *out_of_line_constraint* syntax to define an integrity constraint as part of the table definition.

Note: You must specify a PRIMARY KEY constraint for an index-organized table, and it cannot be DEFERRABLE.

> **See Also:** the syntax description of *out_of_line_constraint* in the *constraints* on page 7-5

physical_properties

The physical properties relate to the treatment of extents and segments and to the storage characteristics of the table.

segment_attributes_clause

physical_attributes_clause

The *physical_attributes_clause* lets you specify the value of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics of the table.

- For a nonpartitioned table, each parameter and storage characteristic you specify determines the actual physical attribute of the segment associated with the table.

- For partitioned tables, the value you specify for the parameter or storage characteristic is the default physical attribute of the segments associated with all partitions specified in this CREATE statement (and in subsequent ALTER TABLE ... ADD PARTITION statements), unless you explicitly override that value in the PARTITION clause of the statement that creates the partition.

If you omit this clause, then Oracle uses the following default values:

- PCTFREE: 10
- PCTUSED: 40
- INITRANS: 1
- MAXTRANS: Depends on data block size

See Also:

- *physical_attributes_clause* on page 7-52 for a full description of the physical attribute parameters
- *storage_clause* on page 7-56 for a description of storage parameters
- "Storage Example" on page 15-66

TABLESPACE

Specify the tablespace in which Oracle creates the table, object table OID index, partition, LOB data segment, LOB index segment, or index-organized table overflow data segment. If you omit TABLESPACE, then Oracle creates that item in the default tablespace of the owner of the schema containing the table.

For heap-organized tables with one or more LOB columns, if you omit the TABLESPACE clause for LOB storage, then Oracle creates the LOB data and index segments in the tablespace where the table is created.

However, for an index-organized table with one or more LOB columns, if you omit TABLESPACE, then the LOB data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.

For nonpartitioned tables, the value specified for TABLESPACE is the actual physical attribute of the segment associated with the table. For partitioned tables, the value specified for TABLESPACE is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and on subsequent ALTER TABLE ... ADD PARTITION statements), unless you specify TABLESPACE in the PARTITION description.

> **See Also:** CREATE TABLESPACE on page 15-80 for more information on tablespaces

logging_clause

Specify whether the creation of the table (and any indexes required because of constraints), partition, or LOB storage characteristics will be logged in the redo log file (`LOGGING`) or not (`NOLOGGING`).The logging attribute of the table is independent of that of its indexes.

This attribute also specifies whether subsequent Direct Loader (SQL*Loader) and direct-path `INSERT` operations against the table, partition, or LOB storage are logged (`LOGGING`) or not logged (`NOLOGGING`).

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

data_segment_compression

Use the *data_segment_compression* clause to instruct Oracle whether to compress data segments to reduce disk use. The `COMPRESS` keyword enables data segment compression. The `NOCOMPRESS` keyword disables data segment compression. `NOCOMPRESS` is the default.

This clause is especially useful in environments such as data warehouses, where the amount of insert and update operations is small. You can specify data segment compression:

- For an entire table heap-organized table (in the *physical_properties* clause of *relational_table* or *object_table*)
- For a range partition (in the *table_partition_description* of the *range_partitioning* clause)
- For a list partition (in the *table_partition_description* of the *list_partitioning* clause)
- For the storage table of a nested table (in the *nested_table_col_properties* clause)

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference* for information on calculating the compression ratio and to *Oracle9i Data Warehousing Guide* for information on data compression usage scenarios

Restrictions on *data_segment_compression*:

- You cannot specify data segment compression for an index-organized table, for any overflow segment or partition of an overflow segment, or for any mapping table segment of an index-organized table.
- You cannot specify data segment compression for hash partitions or for either hash or list subpartitions.
- You cannot specify data segment compression for an external table.

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with `LOGGING` and `NOLOGGING`, respectively. Although `RECOVERABLE` and `UNRECOVERABLE` are supported for backward compatibility, Oracle Corporation strongly recommends that you use the `LOGGING` and `NOLOGGING` keywords.

Restrictions on [UN]RECOVERABLE:

- You cannot specify `RECOVERABLE` for partitioned tables or LOB storage characteristics.
- You cannot specify `UNRECOVERABLE` for a partitioned or index-organized tables.
- You can specify `UNRECOVERABLE` only with `AS` *`subquery`*.

ORGANIZATION

The `ORGANIZATION` clause lets you specify the order in which the data rows of the table are stored.

HEAP `HEAP` indicates that the data rows of *`table`* are stored in no particular order. This is the default.

INDEX `INDEX` indicates that *`table`* is created as an index-organized table. In an index-organized table, the data rows are held in an index defined on the primary key for the table.

EXTERNAL `EXTERNAL` indicates that table is a read-only table located outside the database.

See Also: "External Table Example" on page 15-70

index_org_table_clause

Use the *index_org_table_clause* to create an index-organized table. Oracle maintains the table rows (both primary key column values and nonkey column values) in an index built on the primary key. Index-organized tables are therefore best suited for primary key-based access and manipulation. An index-organized table is an alternative to:

- A nonclustered table indexed on the primary key by using the CREATE INDEX statement
- A clustered table stored in an indexed cluster that has been created using the CREATE CLUSTER statement that maps the primary key for the table to the cluster key

You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row. The primary key cannot be DEFERRABLE. Use the primary key instead of the rowid for directly accessing index-organized rows.

If an index-organized table is partitioned and contains LOB columns, then you should specify the *index_org_table_clause* first, then the *LOB_storage_clause*, and then the appropriate *table_partitioning_clauses*.

See Also: "Index-Organized Table Example" on page 15-70

Note: You cannot use the TO_LOB function to convert a LONG column to a LOB column in the subquery of a CREATE TABLE ...AS SELECT statement if you are creating an index-organized table. Instead, create the index-organized table without the LONG column, and then use the TO_LOB function in an INSERT ... AS SELECT statement.

Restrictions on index-organized tables:

- You cannot specify a column of type ROWID for an index-organized table.
- You cannot specify the *composite_partitioning_clause* for an index-organized table.

PCTTHRESHOLD *integer* Specify the percentage of space reserved in the index block for an index-organized table row. PCTTHRESHOLD must be large enough to hold the primary key. All trailing columns of a row, starting with the column that

causes the specified threshold to be exceeded, are stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. If you do not specify PCTTHRESHOLD, the default is 50.

Restriction: You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

mapping_table_clause Specify MAPPING TABLE to instruct Oracle to create a mapping of local to physical ROWIDs and store them in a heap-organized table. This mapping is needed in order to create a bitmap index on the index-organized table.

Oracle creates the mapping table in the same tablespace as its parent index-organized table. You cannot query, perform DML operations on, or modify the storage characteristics of the mapping table.

Restriction on the *mapping_table_clause*: You cannot specify this clause for a partitioned index-organized table.

key_compression The *key_compression* clauses let you enable or disable key compression for index-organized tables.

- Specify COMPRESS to enable **key compression**, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length (number of prefix columns to compress).

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.

  Restriction on key compression: At the partition level, you can specify COMPRESS, but you cannot specify the prefix length with *integer*.

- Specify NOCOMPRESS to disable key compression in index-organized tables. This is the default.

index_org_overflow_clause The *index_org_overflow_clause* lets you instruct Oracle that index-organized table data rows exceeding the specified threshold are placed in the data segment specified in this clause.

- When you create an index-organized table, Oracle evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, then Oracle raises an error and does not execute the CREATE TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.

- All physical attributes and storage characteristics you specify in this clause after the OVERFLOW keyword apply only to the overflow segment of the table. Physical attributes and storage characteristics for the index-organized table itself, default values for all its partitions, and values for individual partitions must be specified before this keyword.
- If the index-organized table contains one or more LOB columns, then the LOBs will be stored out-of-line unless you specify OVERFLOW, even if they would otherwise be small enough be to stored inline.
- If table is partitioned, then Oracle equipartitions the overflow data segments with the primary key index segments.

INCLUDING *column_name* Specify a column at which to divide an index-organized table row into index and overflow portions. The primary key columns are always stored in the index. *column_name* can be either the last primary-key column or any non-primary-key column. All non-primary-key columns that follow *column_name* are stored in the overflow data segment.

Restriction: You cannot specify this clause for individual partitions of an index-organized table.

Note: If an attempt to divide a row at *column_name* causes the size of the index portion of the row to exceed the PCTTHRESHOLD value (either specified or default), Oracle breaks up the row based on the PCTTHRESHOLD value.

---

supplemental_logging_props

The *supplemental_logging_props* clause lets you instruct Oracle to put additional data into the log stream to support log-based tools.

external_table_clause

Use the *external_table_clause* to create an external table, which is a read-only table whose metadata is stored in the database but whose data in stored outside database. External tables let you query data without first loading it into the database, among other capabilities.

> **See Also:** *Oracle9i Data Warehousing Guide*, *Oracle9i Database Administrator's Guide*, and *Oracle9i Database Utilities* for information on the uses for external tables

Because external tables have no data in the database, you define them with a small subset of the clauses normally available when creating tables.

- Within the *relational_properties* clause, you can specify only *column*, *datatype*, and *inline_constraint*.
- Within the *physical_properties_clause*, you can specify only the organization of the table (ORGANIZATION EXTERNAL *external_table_clause*).
- Within the *table_properties* clause, you can specify only the *parallel_clause*. The *parallel_clause* lets you parallelize subsequent queries on the external data.

Restrictions on external tables:

- No other clauses are permitted in the same CREATE TABLE statement if you specify the *external_table_clause*.
- An external table cannot be a temporary table.
- You cannot specify constraints on an external table.
- An external table cannot have object type columns, LOB columns, or LONG columns.

TYPE

TYPE *access_driver_type* indicates the **access driver** of the external table. The access driver is the API that interprets the external data for the database. If you do not specify TYPE, then Oracle uses the default access driver, ORACLE_LOADER.

> **See Also:** *Oracle9i Database Utilities* for information about the ORACLE_LOADER access driver

DEFAULT DIRECTORY

DEFAULT DIRECTORY lets you specify a default directory object corresponding to a directory on the file system where the external data sources may reside. The default directory can also be used by the access driver to store auxiliary files such as error logs.

ACCESS PARAMETERS

The optional ACCESS PARAMETERS clause lets you assign values to the parameters of the specific access driver for this external table:

- The *opaque_format_spec* lets you list the parameters and their values. Please refer to *Oracle9i Database Utilities* for information on how to specify values for the *opaque_format_spec*.
- USING CLOB *subquery* lets you derive the parameters and their values through a subquery. The subquery cannot contain any set operators or an ORDER BY clause. It must return one row containing a single item of datatype CLOB.

Whether you specify the parameters in an *opaque_format_spec* or derive them using a subquery, Oracle does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

LOCATION

The LOCATION clause lets you specify one or more external data sources. Usually the *location_specifier* is a file, but it need not be. Oracle does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.

REJECT LIMIT

The REJECT LIMIT clause lets you specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is 0.

CLUSTER Clause

The CLUSTER clause indicates that the table is to be part of *cluster*. The columns listed in this clause are the table columns that correspond to the cluster's columns. Generally, the cluster columns of a table are the column or columns that make up its primary key or a portion of its primary key.

See Also: CREATE CLUSTER on page 13-2

Specify one column from the table for each column in the cluster key. The columns are matched by position, not by name.

A clustered table uses the cluster's space allocation. Therefore, do not use the PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameters, the TABLESPACE clause, or the *storage_clause* with the CLUSTER clause.

Restrictions on clustered tables:

- Object tables and tables containing LOB columns cannot be part of a cluster.

- You cannot specify CLUSTER with either ROWDEPENDENCIES or NOROWDEPENDENCIES unless the cluster has been created with the same ROWDEPENDENCIES or NOROWDEPENDENCIES setting.

table_properties

column_properties

Use the *column_properties* clauses to specify the storage attributes of a column.

object_type_col_properties

The *object_type_col_properties* determine storage characteristics of an object column or attribute or an element of a collection column or attribute.

column For *column*, specify an object column or attribute.

substitutable_column_clause The *substitutable_column_clause* indicates whether object columns or attributes in the same hierarchy are substitutable for each other. You can specify that a column is of a particular type, or whether it can contain instances of its subtypes, or both.

- If you specify ELEMENT, you constrain the element type of a collection column or attribute to a subtype of its declared type.
- The IS OF [TYPE] (ONLY *type*) clause constrains the type of the object column to a subtype of its declared type.
- NOT SUBSTITUTABLE AT ALL LEVELS indicates that the object column cannot hold instances corresponding to any of its subtypes. Also, substitution is disabled for any embedded object attributes and elements of embedded nested tables and varrays. The default is SUBSTITUTABLE AT ALL LEVELS.

  Restrictions on the *substitutable_column_clause*:

  - You cannot specify this clause for an attribute of an object column. However, you can specify this clause for a object type column of a relational table, and for an object column of an object table if the substitutability of the object table itself has not been set.
  - For a collection type column, the only part of this clause you can specify is [NOT] SUBSTITUTABLE AT ALL LEVELS.

LOB_storage_clause

The *LOB_storage_clause* lets you specify the storage attributes of LOB data segments.

For a nonpartitioned table (that is, when specified in the *physical_properties* clause without any of the partitioning clauses), this clause specifies the table's storage attributes of LOB data segments.

For a partitioned table, Oracle implements this clause depending on where it is specified:

- For a partitioned table specified at the table level (that is, when specified in the *physical_properties* clause along with one of the partitioning clauses), this clause specifies the default storage attributes for LOB data segments associated with each partition or subpartition. These storage attributes apply to all partitions or subpartitions unless overridden by a *LOB_storage_clause* at the partition or subpartition level.
- For an individual partition of a partitioned table (that is, when specified as part of a *table_partition_description*), this clause specifies the storage attributes of the data segments of the partition or the default storage attributes of any subpartitions of the partition. A partition-level *LOB_storage_clause* overrides a table-level *LOB_storage_clause*.
- For an individual subpartition of a partitioned table (that is, when specified as part of *subpartition_by_hash* or *subpartition_by_list*), this clause specifies the storage attributes of the data segments of the subpartition. A subpartition-level *LOB_storage_clause* overrides both partition-level and table-level *LOB_storage_clauses*.

Restriction on LOB storage in partitioned tables: You cannot specify the *LOB_index_clause* if *table* is partitioned.

LOB_item Specify the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table. Oracle automatically creates a system-managed index for each *LOB_item* you create.

LOB_segname Specify the name of the LOB data segment. You cannot use *LOB_segname* if you specify more than one *LOB_item*.

LOB_parameters The *LOB_parameters* clause lets you specify various elements of LOB storage.

- ENABLE STORAGE IN ROW: If you enable storage in row, then the LOB value is stored in the row (inline) if its length is less than approximately 4000 bytes minus system control information. This is the default.

  Restriction on storage in row: For an index-organized table, you cannot specify this parameter unless you have specified an OVERFLOW segment in the *index_org_table_clause*.

- DISABLE STORAGE IN ROW: If you disable storage in row, then the LOB value is stored out of line (outside of the row) regardless of the length of the LOB value.

  Note: The LOB locator is always stored inline (inside the row) regardless of where the LOB value is stored. You cannot change the value of STORAGE IN ROW once it is set except by moving the table. See the *move_table_clause* of ALTER TABLE on page 11-85.

- CHUNK *integer*: Specify the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, then Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, then Oracle allocates 4096 bytes (2 blocks). The maximum value is 32768 (32K), which is the largest Oracle block size allowed. The default CHUNK size is one Oracle database block.

  You cannot change the value of CHUNK once it is set.

  Note: The value of CHUNK must be less than or equal to the value of NEXT (either the default value or that specified in the *storage_clause*). If CHUNK exceeds the value of NEXT, then Oracle returns an error.

- PCTVERSION *integer*: Specify the maximum percentage of overall LOB storage space used for maintaining old versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until they consume 10% of the overall LOB storage space.

  You can specify the PCTVERSION parameter whether the database is running in manual or automatic undo mode. PCTVERSION is the default in manual undo mode. RETENTION is the default in automatic undo mode.

  Restriction on PCTVERSION: You cannot specify both PCTVERSION and RETENTION.

- RETENTION: Use this clause to indicate that Oracle should retain old versions of this LOB column. Oracle uses the value of the UNDO_RETENTION initialization parameter to determine the amount (in time) of committed undo data to retain in the database.

  You can specify the RETENTION parameter only if the database is running in automatic undo mode. In this mode, RETENTION is the default value unless you specify PCTVERSION.

  Restriction on RETENTION: You cannot specify both PCTVERSION and RETENTION.

- FREEPOOLS *integer*: Specify the number of groups of free lists for the LOB segment. Normally *integer* will be the number of instances in a Real Application Clusters environment or 1 for a single-instance database.

  You can specify this parameter only if the database is running in automatic undo mode. In this mode, FREEPOOLS is the default unless you specify the FREELIST GROUPS parameter of the *storage_clause*. If you specify neither FREEPOOLS nor FREELIST GROUPS, then Oracle uses a default of FREEPOOLS 1 if the database is in automatic undo management mode and a default of FREELIST GROUPS 1 if the database is in manual undo management mode.

  Restriction on FREEPOOLS: You cannot specify both FREEPOOLS and the FREELIST GROUPS parameter of the *storage_clause*.

LOB_index_clause This clause has been deprecated. If you specify this clause, then Oracle ignores it. Oracle automatically generates an index for each LOB column and names and manages the LOB indexes internally.

See Also:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)* for detailed information about LOBs, including guidelines for creating gigabyte LOBs
- "LOB Column Example" on page 15-69
- *Oracle9i Database Migration* for information on how Oracle manages LOB indexes in tables migrated from earlier versions

varray_col_properties

The *varray_col_properties* let you specify separate storage characteristics for the LOB in which a varray will be stored. If *varray_item* is a multilevel collection,

then Oracle stores all collection items nested within *varray_item* in the same LOB in which *varray_item* is stored.

- For a nonpartitioned table (that is, when specified in the *physical_properties* clause without any of the partitioning clauses), this clause specifies the storage attributes of the LOB data segments of the varray.
- For a partitioned table specified at the table level (that is, when specified in the *physical_properties* clause along with one of the partitioning clauses), this clause specifies the default storage attributes for the varray's LOB data segments associated with each partition (or its subpartitions, if any).
- For an individual partition of a partitioned table (that is, when specified as part of a *table_partition_description*), this clause specifies the storage attributes of the varray's LOB data segments of that partition or the default storage attributes of the varray's LOB data segments of any subpartitions of this partition. A partition-level *varray_col_properties* overrides a table-level *varray_col_properties*.
- For an individual subpartition of a partitioned table (that is, when specified as part of *subpartition_by_hash* or *subpartition_by_list*), this clause specifies the storage attributes of the varray's data segments of this subpartition. A subpartition-level *varray_col_properties* overrides both partition-level and table-level *varray_col_properties*.

STORE AS LOB Clause If you specify STORE AS LOB,

- If the maximum varray size is less than 4000 bytes *and* you have not disabled storage in row, then Oracle stores the varray inline.
- If the maximum varray size is greater than 4000 bytes *or* if you have disabled storage in row, then Oracle stores in the varray out of line.

If you do not specify STORE AS LOB, then Oracle handles varray storage differently from other LOBs. Storage is based on the maximum possible size of the varray (that is the number of elements times the element size, plus a small amount for system control information) rather than on the actual size of a varray column.

- If you do not specify this clause, and the maximum size of the varray is less than 4000 bytes, then the varray is not stored as a LOB at all, but as inline raw data.
- If you do not specify this clause, and the maximum size is greater than 4000 bytes, then the varray is always stored as a LOB. If the actual size is less than 4000 bytes, then it will be stored as an inline LOB, and if the actual size is

greater than 4000 bytes, then it will be stored as an out-of-line LOB, as is true for other LOB columns.

Restriction on storing varrays as LOBs: You cannot specify the TABLESPACE parameter of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

substitutable_column_clause The *substitutable_column_clause* has the same behavior as described for *object_type_col_properties* on page 15-36.

> **See Also:** "Substitutable Table and Column Examples" on page 15-67

nested_table_col_properties

The *nested_table_col_properties* let you specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. The storage table is created in the same tablespace as its parent table (using the default storage characteristics) and stores the nested table values of the column for which it was created.

You must include this clause when creating a table with columns or column attributes whose type is a nested table. Clauses within *nested_table_col_properties* that function the same way they function for parent object tables are not repeated here.

nested_item Specify the name of a column (or a top-level attribute of the table's object type) whose type is a nested table.

COLUMN_VALUE If the nested table is a multilevel collection, then the inner nested table or varray may not have a name. In this case, specify COLUMN_VALUE in place of the *nested_item* name.

> **See Also:** "Multi-level Collection Example" on page 15-68 for examples using *nested_item* and COLUMN_VALUE

storage_table Specify the name of the table where the rows of *nested_item* reside. For a nonpartitioned table, the storage table is created in the same schema and the same tablespace as the parent table. For a partitioned table, the storage table is created in the default tablespace of the schema.

Restrictions on *storage_table***:**

- You cannot partition the storage table of a nested table.

- You cannot query or perform DML statements on *storage_table* directly, but you can modify its storage characteristics by specifying its name in an ALTER TABLE statement.

  See Also: ALTER TABLE on page 11-2 for information about modifying nested table column storage characteristics

RETURN AS Specify what Oracle returns as the result of a query.

- VALUE returns a copy of the nested table itself.
- LOCATOR returns a collection locator to the copy of the nested table.

  Note: The locator is scoped to the session and cannot be used across sessions. Unlike a LOB locator, the collection locator cannot be used to modify the collection instance.

If you do not specify the *segment_attributes_clause* or the *LOB_storage_clause*, then the nested table is heap organized and is created with default storage characteristics.

Restrictions on *nested_table_col_properties:*

- You cannot specify this clause for a temporary table.
- You cannot specify the *OID_clause*.
- You cannot specify TABLESPACE (as part of the *segment_attributes_clause*) for a nested table. The tablespace is always that of the parent table.
- At create time, you cannot specify (as part of *object_properties*) an *out_of_line_ref_constraint*, *inline_ref_constraint*, or foreign key constraint for the attributes of a nested table. However, you can modify a nested table to add such constraints using ALTER TABLE.
- You cannot query or perform DML statements on the storage table directly, but you can modify the nested table column storage characteristics by using the name of storage table in an ALTER TABLE statement.

See Also:

- ALTER TABLE on page 11-2 for information about modifying nested table column storage characteristics
- "Nested Table Example" on page 15-68 and "Multi-level Collection Example" on page 15-68

XMLType_column_properties

The *XMLType_column_properties* let you specify storage attributes for an XMLTYPE column.

XMLType_storage XMLType columns can be stored either in LOB or object-relational columns.

- Specify STORE AS OBJECT RELATIONAL if you want Oracle to store the XMLType data in object-relational columns. Storing data object relationally lets you define indexes on the relational columns and enhances query performance.

  If you specify object-relational storage, you must also specify the *XMLSchema_spec* clause.

- Specify STORE AS CLOB if you want Oracle to store the XMLType data in a CLOB column. Storing data in a CLOB column preserves the original content and enhances retrieval time.

  If you specify LOB storage, you can specify either LOB parameters or the *XMLSchema_spec* clause, but not both. Specify the *XMLSchema_spec* clause if you want to restrict the table or column to particular schema-based XML instances.

XMLSchema_spec This clause lets you specify the URL of a registered XMLSchema (in the XMLSCHEMA clause or as part of the ELEMENT clause) and an XML element name. You must specify an element, although the XMLSchema URL is optional. If you do specify an XMLSchema URL, you must already have registered the XMLSchema using the DBMS_XMLSCHEMA package.

See Also:

- *LOB_storage_clause* on page 11-45 for information on the *LOB_segname* and *LOB_parameters* clauses
- "XMLType Column Examples" on page 15-72 for examples of XMLType columns in object-relational tables and "Using XML in SQL Statements" on page D-11 for an example of creating an XMLSchema
- *Oracle9i XML Database Developer's Guide - Oracle XML DB* for more information on XMLType columns and tables and on creating XMLSchemas

table_partitioning_clauses

Use the *table_partitioning_clauses* to create a partitioned table.

Restrictions on partitioning in general:

- You cannot partition a table that is part of a cluster.
- You cannot specify a TIMESTAMP WITH TIME ZONE column as part of the partitioning key.

Note: The storage of partitioned database entities in tablespaces of different block sizes is subject to several restrictions. Please refer to *Oracle9i Database Administrator's Guide* for a discussion of these restrictions.

See Also: "Partitioning Examples" on page 15-73

range_partitioning

Use the *range_partitioning* clause to partition the table on ranges of values from the column list. For an index-organized table, the column list must be a subset of the primary key columns of the table.

column

Specify an ordered list of columns used to determine into which partition a row belongs (the **partitioning key**).

Restriction on *column*: The columns in the column list can be of any built-in datatype except ROWID, LONG, or LOB.

PARTITION *partition*

The name *partition* must conform to the rules for naming schema objects and their part as described in "Schema Object Naming Rules" on page 2-111. If you omit *partition*, then Oracle generates a name with the form SYS_P*n*.

Notes:

- You can specify up to 64K-1 partitions and 64K-1 subpartitions. For a discussion of factors that might impose practical limits less than this number, please refer to *Oracle9i Database Administrator's Guide.*
- You can create a partitioned table with just one partition. However, this is different from a nonpartitioned table. For instance, you cannot add a partition to a nonpartitioned table.

range_values_clause

Specify the noninclusive upper bound for the current partition. The value list is an ordered list of literal values corresponding to the column list in the *range_partitioning* clause. You can substitute the keyword MAXVALUE for any literal in in the value list. MAXVALUE specifies a maximum value that will always sort higher than any other value, including NULL.

Specifying a value other than MAXVALUE for the highest partition bound imposes an implicit integrity constraint on the table.

Note: If *table* is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, then you must use the TO_DATE function with the YYYY 4-character format mask for the year. (The RRRR format mask is not supported.) The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT.

See Also:

- *Oracle9i Database Concepts* for more information about partition bounds
- *Oracle9i Database Globalization Support Guide* for more information on these initialization parameters
- "Range Partitioning Example" on page 15-73

table_partition_description

Use the *table_partition_description* to define the physical and storage characteristics of the table.

The *segment_attributes_clause* and *data_segment_compression* clause have the same function as described for the *table_properties* of the table as a whole.

The *key_compression* clause and OVERFLOW clause have the same function as described for the *index_org_table_clause*.

LOB_storage_clause The *LOB_storage_clause* lets you specify LOB storage characteristics for one or more LOB items in this partition or in any list subpartitions of this partition. If you do not specify the *LOB_storage_clause* for a LOB item, then Oracle generates a name for each LOB data partition. The system-generated names for LOB data and LOB index partitions take the form SYS_LOB_P*n* and SYS_IL_P*n*, respectively, where P stands for "partition" and *n* is a system-generated number. The corresponding system-generated names for LOB subpartitions are SYS_LOB_SUBP*n* and SYS_IL_SUBP*n*.

varray_col_properties The *varray_col_properties* lets you specify storage characteristics for one or more varray items in this partition or in any list subpartitions of this partition.

Restriction on *table_partition_description*: The *partition_level_subpartition* clause is valid only for composite-partitioned tables. See *partition_level_subpartition* on page 15-51.

hash_partitioning

Use the *hash_partitioning* clause to specify that the table is to be partitioned using the hash method.

column Specify an ordered list of columns used to determine into which partition a row belongs (**the partitioning key**).

Oracle assigns rows to partitions using a hash function on values found in columns designated as the partitioning key. You can specify hash partitioning in one of two ways:

- ***individual_hash_partitions***: Use this clause to specify individual partitions by name. If you omit the partition name, then Oracle assigns partition names of the form SYS_P*n*.

  Restriction on *individual_hash_partitions***:** The only clause you can specify in the *partitioning_storage_clause* is the TABLESPACE clause.

  > **Note:** If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

- ***hash_partitions_by_quantity***: Alternatively, you can specify the number of partitions. In this case, Oracle assigns partition names of the form SYS_P*n*. The STORE IN clause specifies one or more tablespaces where the hash partitions are to be stored. The number of tablespaces does not have to equal the number of partitions. If the number of partitions is greater than the number of tablespaces, then Oracle cycles through the names of the tablespaces.

For both methods of hash partitioning, for optimal load balancing you should specify a number of partitions that is a power of 2. Also for both methods of hash partitioning, the only attribute you can specify for hash partitions is TABLESPACE. Hash partitions inherit all other attributes from table-level defaults.

Tablespace storage specified at the table level is overridden by tablespace storage specified at the partition level, which in turn is overridden by tablespace storage specified at the subpartition level.

If you specify tablespace storage in both the STORE IN clause of the *hash_partitions_by_quantity* clause and the TABLESPACE clause of the *partitioning_storage_clause*, then the STORE IN clause determines placement of partitions as the table is being created. The TABLESPACE clause determines the default tablespace at the table level for subsequent operations.

> **See Also:** *Oracle9i Database Concepts* for more information on hash partitioning

Restrictions on hash partitioning:

- You cannot specify more than 16 columns in *column_list*.
- The column list cannot contain the ROWID or UROWID pseudocolumns.
- The column list can be of any built-in datatype except ROWID, LONG, or LOB.

> **See Also:** *Oracle9i Database Globalization Support Guide* for more information on character set support

list_partitioning

Use the *list_partitioning* clause to partition the table on lists of literal values from *column*. List partitioning is useful for controlling how individual rows map to specific partitions.

If you omit the partition name, then Oracle assigns partition names of the form SYS_P*n*.

list_values_clause The *list_values_clause* of each partition must have at least one value. No value (including NULL) can appear in more than one partition. List partitions are not ordered.

Note: If you specify the literal NULL for a partition value in the VALUES clause, then to access data in that partition in subsequent queries, you must use an IS NULL condition in the WHERE clause, rather than a comparison condition.

The DEFAULT keyword creates a partition into which Oracle will insert any row that does not map to another partition. Therefore, you can specify DEFAULT for only one partition, and you cannot specify any other values for that partition. Further, the default partition must be the last partition you define (similar to the use of MAXVALUE for range partitions).

The string comprising the list of values for each partition can be up to 4K bytes. The total number of values for all partitions cannot exceed 64K-1.

Restrictions on list partitioning:

- You cannot subpartition a list partition.
- You can specify only one partitioning key in *column_list*, and it cannot be a LOB column.

- If the partitioning key is an object type column, then you can partition on only one attribute of the column type.
- Each value in the *list_values_clause* must be unique among all partitions of *table*.
- You cannot list partition an index-organized table.

composite_partitioning

Use the *composite_partitioning* clause to first partition *table* by range, and then partition the partitions further into hash or list subpartitions. This combination of range partitioning and hash or list subpartitioning is called **composite partitioning**.

After establishing the type of subpartitioning you want for each composite partition (using the *subpartition_by_hash* or *subpartition_by_list* clause), you must define each of the range partitions.

- You must specify the *range_values_clause*, which has the same requirements as for noncomposite range partitions.
- Use the *table_partition_description* to define the physical and storage characteristics of the each partition. Within the *table_partition_description*, you can use the *partition_level_subpartition* clause to define the properties of individual subpartitions.
- If you omit the partition name, then Oracle generates a name with the form SYS_P*n*.
- The only characteristic you can specify for a hash or list subpartition or a LOB subpartition is TABLESPACE.

Restriction on composite partitioning: You cannot specify composite partitioning for an index-organized table. Therefore, the OVERFLOW clause of the *table_partition_description* is not valid for composite-partitioned tables.

subpartition_template The *subpartition_template* is a common optional element of both range-hash and range-list composite partitioning. The template lets you define default subpartitions for each table partition. Oracle will create these default subpartitions in any partition for which you do not explicitly define subpartitions. This clause is useful for creating symmetric partitions. You can override this clause by explicitly defining subpartitions at the partition level (in the *partition_level_subpartition* clause).

When defining subpartitions with a template, you must specify a name for each subpartition.

Restrictions on the subpartition template:

- The only clause of *partitioning_storage_clause* you can specify is the TABLESPACE clause.
- If you specify TABLESPACE for one LOB subpartition, then you must specify TABLESPACE for all of the LOB subpartitions of that LOB column. You can specify the same tablespace for more than one subpartition.
- If you specify separate LOB storage for list subpartitions using the *partitioning_storage_clause*, either in the *subpartition_template* or when defining individual subpartitions, then you must specify *LOB_segname* (for both LOB and varray columns).

subpartition_by_hash

Use the *subpartition_by_hash* clause to indicate that Oracle should subpartition by hash each partition in *table*. The subpartitioning column list is unrelated to the partitioning key, but is subject to the same restrictions (see *column* on page 15-44).

You can define the subpartitions using the *subpartition_template* or the SUBPARTITIONS *quantity* clause. See *subpartition_template* on page 15-49. In either case, for optimal load balancing you should specify a number of partitions that is a power of 2.

SUBPARTITIONS *quantity* Specify the default number of subpartitions in each partition of *table*, and optionally one or more tablespaces in which they are to be stored.

The default value is 1. If you omit both this clause and *subpartition_template*, then Oracle will create each partition with one hash subpartition unless you subsequently specify the *partition_level_subpartition* clause.

Restriction on hash subpartitioning: In addition to the restrictions for composite partitioning in general (see *composite_partitioning* on page 15-49), for hash subpartitioning in *subpartition_template*, you cannot specify the *list_values_clause*.

subpartition_by_list

Use the *subpartition_by_list* clause to indicate that Oracle should subpartition each partition in *table* by literal values from *column*.

If you omit *subpartition_template*, then you can define list subpartitions individually for each partition using the *partition_level_subpartition* clause of *table_partition_description*. If you omit both *subpartition_template* and *partition_level_subpartition*, then Oracle creates a single DEFAULT subpartition.

Restrictions on list subpartitioning: In addition to the restrictions for composite partitioning in general (see *composite_partitioning* on page 15-49), for list subpartitioning:

- You can specify only one subpartitioning key column.
- You must specify the *list_values_clause*, which is subject to the same requirements as at the table level.
- In the *subpartition_template*, you cannot specify the *hash_subpartition_quantity* clause.

partition_level_subpartition

This clause of the *table_partition_description* is valid *only* for composite-partitioned tables. This clause lets you specify hash or list subpartitions for *partition*. This clause overrides the default settings established in the *subpartition_by_hash* clause (for range-hash composite partitions) or in the *subpartition_template* (for range-hash or range-list composite partitions).

For all composite partitions:

- You can specify the number of subpartitions (and optionally one or more tablespaces where they are to be stored). In this case, Oracle assigns subpartition names of the form SYS_SUBPn. The number of tablespaces does not have to equal the number of subpartitions. If the number of partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.
- Alternatively, you can use the *subpartition_spec* to specify individual subpartitions by name, and optionally the tablespace where each should be stored.
- If you omit *partition_level_subpartition* and if you have created a subpartition template, Oracle uses the template to create subpartitions. If you have not created a subpartition template, Oracle creates one hash subpartition or one DEFAULT list subpartition.
- If you omit *partition_level_subpartition* entirely, Oracle assigns subpartition names as follows:

  - If you have specified a subpartition template *and* you have specified partition names, then Oracle generates subpartition names of the form "*partition_name* underscore (_) *subpartition_name*" (for example, P1_SUB1).
  - If you have not specified a subpartition template *or* if you have specified a subpartition template but did not specify partition names, then Oracle generates subpartition names of the form SYS_SUBP*n*.
- In *partition_spec*, the only clause of the *partitioning_storage_clause* you can specify is the TABLESPACE clause.

For range-hash composite partitions, the *list_values_clause* of *subpartition_spec* is not relevant and is invalid.

For range-list composite partitions:

- The *hash_subpartition_quantity* is not relevant, so you must use the lower branch of *partition_level_subpartition*.
- Within *subpartition_spec*, you must specify the *list_values_clause* for each subpartition, and the values you specify for each subpartition cannot exist in any other subpartition of the same partition.

object_table

The OF clause lets you explicitly create an **object table** of type *object_type*. The columns of an object table correspond to the top-level attributes of type *object_type*. Each row will contain an object instance, and each instance will be assigned a unique, system-generated object identifier (OID) when a row is inserted. If you omit *schema*, then Oracle creates the object table in your own schema.

Object tables (as well as XMLType tables, object views, and XMLType views) do not have any column names specified for them. Therefore, Oracle defines a system-generated column SYS_NC_ROWINFO$. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

See Also: "Object Column and Table Examples" on page 15-77

object_table_substitution

Use the *object_table_substitution* clause to specify whether row objects corresponding to subtypes can be inserted into this object table.

NOT SUBSTITUTABLE AT ALL LEVELS NOT SUBSTITUTABLE AT ALL LEVELS indicates that the object table being created is not substitutable. In addition,

substitution is disabled for all embedded object attributes and elements of embedded nested tables and arrays. The default is SUBSTITUTABLE AT ALL LEVELS.

object_properties

The properties of object tables are essentially the same as those of relational tables. However, instead of specifying columns, you specify attributes of the object.

For *attribute*, specify the qualified column name of an item in an object.

ON COMMIT

The ON COMMIT clause is relevant only if you are creating a temporary table. This clause specifies whether the data in the temporary table persists for the duration of a transaction or a session.

DELETE ROWS Specify DELETE ROWS for a transaction-specific temporary table (this is the default). Oracle will truncate the table (delete all its rows) after each commit.

PRESERVE ROWS Specify PRESERVE ROWS for a session-specific temporary table. Oracle will truncate the table (delete all its rows) when you terminate the session.

OID_clause

The *OID_clause* lets you specify whether the object identifier (OID) of the object table should be system generated or should be based on the primary key of the table. The default is SYSTEM GENERATED.

Restrictions on the *OID_clause*:

- You cannot specify OBJECT IDENTIFIER IS PRIMARY KEY unless you have already specified a PRIMARY KEY constraint for the table.
- You cannot specify this clause for a nested table.

Note: A primary key OID is locally (but not necessarily globally) unique. If you require a globally unique identifier, then you must ensure that the primary key is globally unique.

OID_index_clause

This clause is relevant only if you have specified the *OID_clause* as SYSTEM GENERATED. It specifies an index, and optionally its storage characteristics, on the hidden object identifier column.

For *index*, specify the name of the index on the hidden system-generated object identifier column. If you omit *index*, then Oracle generates a name.

row_movement_clause

The *row_movement_clause* lets you specify whether Oracle can move a table row. It is possible for a row to move, for example, during data segment compression or an update operation on partitioned data.

> **Caution:** If you need static rowids for data access, do not enable row movement. For a normal (heap-organized) table, moving a row changes that row's rowid. For a moved row in an index-organized table, the logical rowid remains valid, although the physical guess component of the logical rowid becomes inaccurate.

- Specify ENABLE to allow Oracle to move a row, thus changing the rowid.
- Specify DISABLE if you want to prevent Oracle from moving a row, thus preventing a change of rowid.

Restriction on *row_movement_clause*: You cannot specify this clause for a nonpartitioned index-organized table.

If you omit this clause, then Oracle disables row movement.

CACHE | NOCACHE | CACHE READS

CACHE Clause For data that is accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the *most recently used* end of the least recently used (LRU) list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

As a parameter in the *LOB_storage_clause*, CACHE specifies that Oracle places LOB data values in the buffer cache for faster access.

Restriction: You cannot specify CACHE for an index-organized table. However, index-organized tables implicitly provide CACHE behavior.

NOCACHE Clause For data that is not accessed frequently, this clause indicates that the blocks retrieved for this table are placed at the *least recently used* end of the LRU list in the buffer cache when a full table scan is performed.

As a parameter in the *LOB_storage_clause*, NOCACHE specifies that the LOB value is either not brought into the buffer cache or brought into the buffer cache and placed at the least recently used end of the LRU list. (The latter is the default behavior.) NOCACHE is the default for LOB storage.

Restriction: You cannot specify NOCACHE for index-organized tables.

CACHE READS CACHE READS applies only to LOB storage. It specifies that LOB values are brought into the buffer cache only during read operations, but not during write operations.

> **See Also:** *logging_clause* on page 7-45 for a description of the *logging_clause* when specified as part of *LOB_parameters*

NOROWDEPENDENCIES | ROWDEPENDENCIES

This clause lets you specify whether *table* will use **row-level dependency tracking**. With this feature, each row in the table has a system change number (SCN) that represents a time greater than or equal to the commit time of the last transaction that modified the row. You cannot change this setting after *table* is created.

ROWDEPENDENCIES Specify ROWDEPENDENCIES if you want to enable row-level dependency tracking. This setting is useful primarily to allow for parallel propagation in replication environments. It increases the size of each row by 6 bytes.

NOROWDEPENDENCIES Specify NOROWDEPENDENCIES if you do not want *table* to use the row level dependency tracking feature. This is the default.

> **See Also:** *Oracle9i Replication* for information about the use of row-level dependency tracking in replication environments

MONITORING | NOMONITORING

MONITORING Specify MONITORING if you want modification statistics to be collected on this table. These statistics are estimates of the number of rows affected by DML statements over a particular period of time. They are available for use by the optimizer or for analysis by the user.

Restriction on MONITORING: You cannot specify MONITORING for a temporary table.

NOMONITORING Specify NOMONITORING if you do not want Oracle to collect modification statistics on the table. This is the default.

Restriction on NOMONITORING: You cannot specify NOMONITORING for a temporary table.

parallel_clause

The *parallel_clause* lets you parallelize creation of the table and set the default degree of parallelism for queries and DML (INSERT, UPDATE, DELETE, and MERGE) operations on the table after creation.

> **Note:** The syntax of the *parallel_clause* supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior than that documented.

NOPARALLEL Specify NOPARALLEL for serial execution. This is the default.

PARALLEL Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* Specification of *integer* indicates the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Notes on the *parallel_clause*

- Parallelism is disabled for tables on which you have defined a trigger or referential integrity constraint.
- If you define a bitmap index on table:
  - If *table* is nonpartitioned, then subsequent DML operations are executed serially.
  - If *table* is partitioned, then Oracle limits the degree of parallelism to the number of partitions accessed in the DML operation.

- If *table* contains any columns of LOB or user-defined object type, then subsequent INSERT, UPDATE, or DELETE operations that modify the LOB or object type column are executed serially without notification. Subsequent queries, however, will be executed in parallel.
- A parallel hint overrides the effect of the *parallel_clause*.
- DML statements and CREATE TABLE ... AS SELECT statements that reference remote objects can run in parallel. However, the "remote object" must really be on a remote database. The reference cannot loop back to an object on the local database (for example, by way of a synonym on the remote database pointing back to an object on the local database).
- DML operations on tables with LOB columns can be parallelized. However, intra-partition parallelism is not supported.

> **See Also:** *Oracle9i Database Performance Tuning Guide and Reference*, *Oracle9i Database Concepts*, *Oracle9i Data Warehousing Guide* for more information on parallelized operations, and "PARALLEL Example" on page 15-67

enable_disable_clause

The *enable_disable_clause* lets you specify whether Oracle should apply a constraint. By default, constraints are created in ENABLE VALIDATE state.

Restrictions on enabling and disabling constraints:

- To enable or disable any integrity constraint, you must have defined the constraint in this or a previous statement.
- You cannot enable a foreign key constraint unless the referenced unique or primary key constraint is already enabled.

> **See Also:** *constraints* on page 7-5 for more information on constraints, "ENABLE VALIDATE Example" on page 15-68, and "DISABLE Example" on page 15-68

ENABLE Clause Specify ENABLE if you want the constraint to be applied to the data in the table.

Note: If you enable a unique or primary key constraint, and if no index exists on the key, Oracle creates a unique index. This index is dropped if the constraint is subsequently disabled, and Oracle rebuilds the index every time the constraint is enabled.

To avoid rebuilding the index and eliminate redundant indexes, create new primary key and unique constraints initially disabled. Then create (or use existing) nonunique indexes to enforce the constraint. Oracle does not drop a nonunique index when the constraint is disabled, so subsequent ENABLE operations are facilitated.

- ENABLE VALIDATE specifies that all old and new data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If any row in the table violates the integrity constraint, the constraint remains disabled and Oracle returns an error. If all rows comply with the constraint, Oracle enables the constraint. Subsequently, if new data violates the constraint, Oracle does not execute the statement and returns an error indicating the integrity constraint violation.

  Note: If you place a primary key constraint in ENABLE VALIDATE mode, the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before entering data into the column and before enabling the table's primary key constraint.

- ENABLE NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint. This clause does not ensure that existing data in the table complies with the constraint and therefore does not require a table lock.

If you specify neither VALIDATE nor NOVALIDATE, the default is VALIDATE.

If you change the state of any single constraint from ENABLE NOVALIDATE to ENABLE VALIDATE, the operation can be performed in parallel, and does not block reads, writes, or other DDL operations.

Restriction on the ENABLE Clause: You cannot enable a foreign key that references a disabled unique or primary key.

DISABLE Clause Specify DISABLE to disable the integrity constraint. Disabled integrity constraints appear in the data dictionary along with enabled constraints. If you do not specify this clause when creating a constraint, Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, because it lets you load large amounts of data while also saving space by not having an index. This setting lets you load data from a nonpartitioned table into a partitioned table using the *exchange_partition_clause* of the ALTER TABLE statement or using SQL*Loader. All other modifications to the table (inserts, updates, and deletes) by other SQL statements are disallowed.

  See Also: *Oracle9i Data Warehousing Guide* for more information on using this setting

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated).

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DISABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

  See Also: *Oracle9i Database Performance Tuning Guide and Reference* for information on when to use this setting

If you specify neither VALIDATE nor NOVALIDATE, the default is NOVALIDATE.

If you disable a unique or primary key constraint that is using a unique index, Oracle drops the unique index.

UNIQUE The UNIQUE clause lets you enable or disable the unique constraint defined on the specified column or combination of columns.

PRIMARY KEY The PRIMARY KEY clause lets you enable or disable the table's primary key constraint.

CONSTRAINT The CONSTRAINT clause lets you enable or disable the integrity constraint named *constraint*.

KEEP | DROP INDEX This clause lets you either preserve or drop the index Oracle has been using to enforce a unique or primary key constraint.

Restriction on KEEP | DROP INDEX: You can specify this clause only when disabling a unique or primary key constraint.

using_index_clause

The *using_index_clause* lets you specify an index for Oracle to use to enforce a unique or primary key constraint, or lets you instruct Oracle to create the index used to enforce the constraint.

You can specify the *using_index_clause* only when enabling unique or primary key constraints. You can specify the clauses of the *using_index_clause* in any order, but you can specify each clause only once.

- If you specify *schema.index*, Oracle attempts to enforce the constraint using the specified index. If Oracle cannot find the index or cannot use the index to enforce the constraint, Oracle returns an error.
- If you specify the *create_index_statement*, Oracle attempts to create the index and use it to enforce the constraint. If Oracle cannot create the index or cannot use the index to enforce the constraint, Oracle returns an error.
- If you neither specify an existing index nor create a new index, Oracle creates the index. In this case:
  - The index receives the same name as the constraint.
  - You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, PCTFREE, and STORAGE parameters for the index. You cannot specify PCTUSED or the *logging_clause* for the index.
  - If *table* is partitioned, you can specify a locally or globally partitioned index for the unique or primary key constraint.

    See Also: "Explicit Index Control Example" on page 7-35 for an example of how you can create an index for Oracle to use in enforcing a constraint

Restrictions on the *using_index_clause*:

- You cannot specify this clause for a view constraint.
- You cannot specify this clause for a NOT NULL, foreign key, or check constraint.

- You cannot specify an index (*schema.index*) or create an index (*create_index_statement*) when enabling the primary key of an index-organized table.

  See Also:

  - *constraints* on page 7-5 for additional information on the *using_index_clause* and on PRIMARY KEY and UNIQUE constraints
  - CREATE INDEX on page 13-62 for a description of LOCAL and the *global_index_clause*, and for a description of NOSORT and the *logging_clause* in relation to indexes
  - *segment_attributes_clause* on page 15-27 for information on the INITRANS, MAXTRANS, TABLESPACE, STORAGE, and PCTFREE parameters

global_partitioned_index The *global_partitioned_index* clause lets you specify that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes. Oracle will partition the global index on the ranges of values from the table columns you specify in *column_list*. You cannot specify this clause for a local index.

The *column_list* must specify a left prefix of the index column list. That is, if the index is defined on columns a, b, and c, then for *column_list* you can specify (a, b, c), or (a, b), or (a, c), but you cannot specify (b, c) or (c) or (b, a).

Restrictions on *column_list*:

- You cannot specify more than 32 columns in *column_list*.
- The columns cannot contain the ROWID pseudocolumn or a column of type ROWID.

  Note: If your enterprise has or will have databases using different character sets, use caution when partitioning on character columns. The sort sequence of characters is not identical in all character sets.

  See Also: *Oracle9i Database Globalization Support Guide* for more information on character set support

index_partitioning_clause Use this clause to describe the individual index partitions. The number of repetitions of this clause determines the number of partitions. If you omit *partition*, Oracle generates a name with the form SYS_P*n*.

For VALUES LESS THAN (*value_list*), specify the (noninclusive) upper bound for the current partition in a global index. The value list is a comma-delimited, ordered list of literal values corresponding to the column list in the *global_partitioned_index* clause. Always specify MAXVALUE as the value of the last partition.

Note: If the index is partitioned on a DATE column, and if the date format does not specify the first two digits of the year, you must use the TO_DATE function with a 4-character format mask for the year. The date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT.

CASCADE Specify CASCADE to disable any integrity constraints that depend on the specified integrity constraint. To disable a primary or unique key that is part of a referential integrity constraint, you must specify this clause.

Restriction on CASCADE: You can specify CASCADE only if you have specified DISABLE.

AS *subquery*

Specify a subquery to determine the contents of the table. The rows returned by the subquery are inserted into the table upon its creation.

For object tables, *subquery* can contain either one expression corresponding to the table type, or the number of top-level attributes of the table type.

> **See Also:** SELECT on page 18-4

If *subquery* returns (in part or totally) the equivalent of an existing materialized view, then Oracle may use the materialized view (for query rewrite) in place of one or more tables specified in *subquery*.

> **See Also:** *Oracle9i Data Warehousing Guide* for more information on materialized views and query rewrite

Oracle derives datatypes and lengths from the subquery. Oracle follows the following rules for integrity constraints and other column and table attributes:

- Oracle automatically defines any NOT NULL constraints on columns in the new table that were explicitly created on the corresponding columns of the selected table if the subquery selects the column rather than an expression containing the column. If any rows violate the constraint, then Oracle does not create the table and returns an error.
- NOT NULL constraints that were implicitly created by Oracle on columns of the selected table (for example, for primary keys) are not carried over to the new table.
- In addition, primary keys, unique keys, foreign keys, check constraints, partitioning criteria, indexes, and column default values are not carried over to the new table.

If all expressions in *subquery* are columns, rather than expressions, then you can omit the columns from the table definition entirely. In this case, the names of the columns of table are the same as the columns in *subquery*.

You can use *subquery* in combination with the TO_LOB function to convert the values in a LONG column in another table to LOB values in a column of the table you are creating.

See Also:

- *Oracle9i Database Migration* for a discussion of why and when to copy LONGs to LOBs
- "Conversion Functions" on page 6-5 for a description of how to use the TO_LOB function
- SELECT on page 18-4 for more information on the *order_by_clause*

parallel_clause If you specify the *parallel_clause* in this statement, then Oracle will ignore any value you specify for the INITIAL storage parameter, and will instead use the value of the NEXT parameter.

See Also: *storage_clause* on page 7-56 for information on these parameters

ORDER BY The ORDER BY clause lets you order rows returned by the subquery.

Note: When specified with CREATE TABLE, this clause does not necessarily order data cross the entire table. (For example, it does not order across partitions.) Specify this clause if you intend to create an index on the same key as the ORDER BY key column. Oracle will cluster data on the ORDER BY key so that it corresponds to the index key.

Restrictions on *subquery*:

- The number of columns in the table must equal the number of expressions in the subquery.
- The column definitions can specify only column names, default values, and integrity constraints, not datatypes.
- You cannot define a foreign key constraint in a CREATE TABLE statement that contains AS *subquery*. Instead, you must create the table without the constraint and then add it later with an ALTER TABLE statement.

XMLType_table

Use the *XMLType_table* syntax to create a table of datatype XMLType.

Object tables (as well as XMLType tables, object views, and XMLType views) do not have any column names specified for them. Therefore, Oracle defines a system-generated column SYS_NC_ROWINFO$. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

XMLType_storage

This clause lets you determine how Oracle manages the storage of the underlying columns.

OBJECT RELATIONAL Specify OBJECT RELATIONAL if you want Oracle to store the XMLType data in object relational columns. If you specify OBJECT RELATIONAL, then you must also specify an XMLSchema in the *XMLSchema_storage_clause*, and you must already have registered the schema (using the DBMS_XMLSCHEMA package). Oracle will create the table conforming to the registered schema.

CLOB Specify CLOB if you want Oracle to store the XML data in a CLOB column. If you specify CLOB, then you may also specify either a LOB segment name, or the *LOB_parameters* clause, or both.

XMLSchema_spec

This clause lets you specify the URL of a registered XMLSchema (in the XMLSCHEMA clause or as part of the ELEMENT clause) and an XML element name. You must specify an element, although the XMLSchema URL is optional. If you do specify an XMLSchema URL, you must already have registered the XMLSchema using the DBMS_XMLSCHEMA package.

See Also:

- *Oracle9i Supplied PL/SQL Packages and Types Reference* for information on the DBMS_XMLSCHEMA package
- *Oracle9i XML Database Developer's Guide - Oracle XML DB* for information on creating and working with XML data
- "XMLType Table Examples" on page 15-71

Examples

General Examples

To statement shows how the employees table owned by the sample Human Resources (hr) schema was created:

```
CREATE TABLE employees_demo
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
        CONSTRAINT emp_last_name_nn NOT NULL
    , email          VARCHAR2(25)
        CONSTRAINT emp_email_nn     NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE  DEFAULT SYSDATE
        CONSTRAINT emp_hire_date_nn  NOT NULL
    , job_id         VARCHAR2(10)
       CONSTRAINT     emp_job_nn  NOT NULL
    , salary         NUMBER(8,2)
       CONSTRAINT     emp_salary_nn  NOT NULL
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , dn             VARCHAR2(300)
    , CONSTRAINT     emp_salary_min
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk
```

```
                    UNIQUE (email)
    ) ;
```

This table contains twelve columns. The `employee_id` column is of datatype `NUMBER`. The `hire_date` column is of datatype `DATE` and has a default value of `SYSDATE`. The `last_name` column is of type `VARCHAR2` and has a `NOT NULL` constraint, and so on.

Storage Example To define the same employees_demo table in the `demo` tablespace with a small storage capacity and limited allocation potential, issue the following statement:

```
CREATE TABLE employees_demo
    ( employee_id    NUMBER(6)
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
         CONSTRAINT emp_last_name_nn NOT NULL
    , email          VARCHAR2(25)
         CONSTRAINT emp_email_nn     NOT NULL
    , phone_number   VARCHAR2(20)
    , hire_date      DATE  DEFAULT SYSDATE
         CONSTRAINT emp_hire_date_nn  NOT NULL
    , job_id         VARCHAR2(10)
       CONSTRAINT     emp_job_nn  NOT NULL
    , salary         NUMBER(8,2)
       CONSTRAINT     emp_salary_nn  NOT NULL
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
    , dn             VARCHAR2(300)
    , CONSTRAINT     emp_salary_min
                     CHECK (salary > 0)
    , CONSTRAINT     emp_email_uk
                     UNIQUE (email)
    )
   TABLESPACE demo
   STORAGE (INITIAL     6144
            NEXT        6144
            MINEXTENTS     1
            MAXEXTENTS     5 );
```

Temporary Table Example The following statement creates a temporary table `today_sales` for use by sales representatives in the sample database. Each sales

representative session can store its own sales data for the day in the table. The temporary data is deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE today_sales
   ON COMMIT PRESERVE ROWS
   AS SELECT * FROM orders WHERE order_date = SYSDATE;
```

Substitutable Table and Column Examples The following statement creates a substitutable table from the `person_t` type, which was created in "Type Hierarchy Example" on page 16-22:

```
CREATE TABLE persons OF person_t;
```

The following statement creates a table with a substitutable column of type `person_t`:

```
CREATE TABLE books (title VARCHAR2(100), author person_t);
```

When you insert into `persons` or `books`, you can specify values for the attributes of `person_t` or any of its subtypes. Example insert statements appear in "Inserting into a Substitutable Tables and Columns: Examples" on page 17-68.

You can extract data from such tables using built-in functions and conditions. For examples, see the functions TREAT on page 6-188 and SYS_TYPEID on page 6-161, and "IS OF type Conditions" on page 5-19.

PARALLEL Example The following statement creates a table using an optimum number of parallel execution servers to scan `employees` and to populate `dept_80`:

```
CREATE TABLE dept_80
   PARALLEL
   AS SELECT * FROM employees
   WHERE department_id = 80;
```

Using parallelism speeds up the creation of the table because Oracle uses parallel execution servers to create the table. After the table is created, querying the table is also faster, because the same degree of parallelism is used to access the table.

NOPARALLEL Example The following statement creates the same table serially. Subsequent DML and queries on the table will also be serially executed.

```
CREATE TABLE dept_80
   AS SELECT * FROM employees
   WHERE department_id = 80;
```

ENABLE VALIDATE Example The following statement shows how the sample table `departments` was created. The example defines a `NOT NULL` constraint, and places it in `ENABLE VALIDATE` state:

```
CREATE TABLE departments_demo
    ( department_id    NUMBER(4)
    , department_name  VARCHAR2(30)
          CONSTRAINT  dept_name_nn  NOT NULL
    , manager_id       NUMBER(6)
    , location_id      NUMBER(4)
    , dn               VARCHAR2(300)
    ) ;
```

DISABLE Example The following statement creates the same `departments_demo` table but also defines a disabled primary key constraint:

```
CREATE TABLE departments_demo
    ( department_id    NUMBER(4)   PRIMARY KEY DISABLE
    , department_name  VARCHAR2(30)
          CONSTRAINT  dept_name_nn  NOT NULL
    , manager_id       NUMBER(6)
    , location_id      NUMBER(4)
    , dn               VARCHAR2(300)
    ) ;
```

Nested Table Example The following statement shows how the sample table `pm.print_media` was created with a nested table column `ad_textdocs_ntab`:

```
CREATE TABLE print_media
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    , press_release     LONG
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab;
```

Multi-level Collection Example The following example shows how an account manager might create a table of customers using two levels of nested tables:

```
CREATE TYPE phone AS OBJECT (telephone NUMBER);
```

```
/
CREATE TYPE phone_list AS TABLE OF phone;
/
CREATE TYPE my_customer AS OBJECT (
   cust_name VARCHAR2(25),
   phones phone_list);
/
CREATE TYPE customer_list AS TABLE OF my_customer;
/
CREATE TABLE business_contacts (
   company_name VARCHAR2(25),
   company_reps customer_list)
   NESTED TABLE company_reps STORE AS outer_ntab
   (NESTED TABLE phones STORE AS inner_ntab);
```

The following variation of this example shows how to use the COLUMN_VALUE keyword if the inner nested table has no column or attribute name:

```
CREATE TYPE phone AS TABLE OF NUMBER;
/
CREATE TYPE phone_list AS TABLE OF phone;
/
CREATE TABLE my_customers (
   name VARCHAR2(25),
   phone_numbers phone_list)
   NESTED TABLE phone_numbers STORE AS outer_ntab
   (NESTED TABLE COLUMN_VALUE STORE AS inner_ntab);
```

LOB Column Example The following statement is a variation of the statement that created the pm.print_media table with some added LOB storage characteristics:

```
CREATE TABLE print_media_new
    ( product_id        NUMBER(6)
    , ad_id             NUMBER(6)
    , ad_composite      BLOB
    , ad_sourcetext     CLOB
    , ad_finaltext      CLOB
    , ad_fltextn        NCLOB
    , ad_textdocs_ntab  textdoc_tab
    , ad_photo          BLOB
    , ad_graphic        BFILE
    , ad_header         adheader_typ
    , press_release     LONG
    ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab_new
    LOB (ad_sourcetext, ad_finaltext) STORE AS
```

```
      (TABLESPACE demo
       STORAGE (INITIAL 6144 NEXT 6144)
       CHUNK 4000
       NOCACHE LOGGING);
```

In the example, Oracle rounds the value of CHUNK up to 4096 (the nearest multiple of the block size of 2048).

Index-Organized Table Example The following statement shows how the sample table hr.countries, which is index organized, was created:

```
CREATE TABLE countries
    ( country_id      CHAR(2)
      CONSTRAINT  country_id_nn NOT NULL
    , country_name    VARCHAR2(40)
    , currency_name   VARCHAR2(25)
    , currency_symbol VARCHAR2(3)
    , region          VARCHAR2(15)
    ,  CONSTRAINT     country_c_id_pk
                     PRIMARY KEY (country_id)
    )
    ORGANIZATION INDEX
    INCLUDING   country_name
    PCTTHRESHOLD 2
    STORAGE
     ( INITIAL  4K
      NEXT  2K
      PCTINCREASE 0
      MINEXTENTS 1
      MAXEXTENTS 1 )
   OVERFLOW
    STORAGE
      ( INITIAL  4K
        NEXT  2K
        PCTINCREASE 0
        MINEXTENTS 1
        MAXEXTENTS 1 );
```

External Table Example The following statement creates an external table that represents a subset of the sample table hr.employees. The *opaque_format_spec* is shown in italics. Please refer to *Oracle9i Database Utilities* for information on the ORACLE_LOADER access driver and how to specify values for the *opaque_format_spec*.

```
CREATE TABLE emps_external (
```

```
   employee_id    NUMBER(6),
   last_name      VARCHAR2(20),
   email          VARCHAR2(25),
   hire_date      DATE,
   job_id         VARCHAR2(10),
   salary         NUMBER(8,2)
)
ORGANIZATION EXTERNAL
(TYPE oracle_loader
 DEFAULT DIRECTORY admin
 ACCESS PARAMETERS
 (
  RECORDS DELIMITED BY newline
  BADFILE 'ulcase1.bad'
  DISCARDFILE 'ulcase1.dis'
  LOGFILE 'ulcase1.log'
  SKIP 20
  FIELDS TERMINATED BY ","  OPTIONALLY ENCLOSED BY '"'
  (
   deptno     INTEGER EXTERNAL,
   dname      CHAR,
   loc        CHAR
  )
 )
 LOCATION ('ulcase1.dat')
)
REJECT LIMIT UNLIMITED;
```

> **See Also:** "Creating a Directory: Examples" on page 13-48 to see how the `admin` directory was created

XMLType Examples

This section contains brief examples of creating an `XMLType` table or `XMLType` column. For a more expanded version of these examples, please refer to "Using XML in SQL Statements" on page D-11.

XMLType Table Examples The following example creates a very simple `XMLType` table with one implicit `CLOB` column:

```
CREATE TABLE xwarehouses OF XMLTYPE;
```

Because Oracle implicitly stores the data in a `CLOB` column, it is subject to all of the restrictions on LOB columns. To avoid these restrictions, you can create an XMLSchema-based table, as shown in the following example. The XMLSchema

must already have been created (see "Using XML in SQL Statements" on page D-11 for more information):

```
CREATE TABLE xwarehouses OF XMLTYPE
   XMLSCHEMA "http://www.oracle.com/xwarehouses.xsd"
   ELEMENT "Warehouse";
```

You can define constraints on an XMLSchema-based table, and you can also create indexes on XMLSchema-based tables, which greatly enhance subsequent queries. You can create object-relational views on XMLType tables, and you can create XMLType views on object-relational tables.

See Also:

- "Using XML in SQL Statements" on page D-11 for an example of adding a constraint
- "Create an Index on an XMLType Table: Example" on page 13-84 for an example of creating an index
- "Creating an XMLType View: Example" on page 16-53 for an example of creating an XMLType view

XMLType Column Examples The following example creates a table with an XMLType column stored as a CLOB. This table does not require an XMLSchema, so the content structure is not predetermined:

```
CREATE TABLE xwarehouses (
   warehouse_id        NUMBER,
   warehouse_spec      XMLTYPE)
   XMLTYPE warehouse_spec STORE AS CLOB
   (TABLESPACE demo
    STORAGE (INITIAL 6144 NEXT 6144)
    CHUNK 4000
    NOCACHE LOGGING);
```

The following example creates a similar table, but stores the XMLType data in an object relational XMLType column whose structure is determined by the specified schema:

```
CREATE TABLE xwarehouses (
   warehouse_id    NUMBER,
   warehouse_spec  XMLTYPE)
   XMLTYPE warehouse_spec STORE AS OBJECT RELATIONAL
      XMLSCHEMA "http://www.oracle.com/xwarehouses.xsd"
      ELEMENT "Warehouse";
```

Partitioning Examples

Range Partitioning Example The `sales` table in the sample schema `sh` is partitioned by range. The following example shows an abbreviated variation of the `sales` table (constraints and storage elements have been omitted from the example):

```
CREATE TABLE range_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold         NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
  (PARTITION SALES_Q1_1998 VALUES LESS THAN (TO_DATE('01-APR-1998','DD-MON-YYYY')),
   PARTITION SALES_Q2_1998 VALUES LESS THAN (TO_DATE('01-JUL-1998','DD-MON-YYYY')),
   PARTITION SALES_Q3_1998 VALUES LESS THAN (TO_DATE('01-OCT-1998','DD-MON-YYYY')),
   PARTITION SALES_Q4_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
   PARTITION SALES_Q1_1999 VALUES LESS THAN (TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION SALES_Q2_1999 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION SALES_Q3_1999 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION SALES_Q4_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
   PARTITION SALES_Q1_2000 VALUES LESS THAN (TO_DATE('01-APR-2000','DD-MON-YYYY')),
   PARTITION SALES_Q2_2000 VALUES LESS THAN (TO_DATE('01-JUL-2000','DD-MON-YYYY')),
   PARTITION SALES_Q3_2000 VALUES LESS THAN (TO_DATE('01-OCT-2000','DD-MON-YYYY')),
   PARTITION SALES_Q4_2000 VALUES LESS THAN (MAXVALUE))
;
```

For information about partitioned table maintenance operations, see the *Oracle9i Database Administrator's Guide.*

List Partitioning Example The following statement shows how the sample table `oe.customers` might have been created as a list-partitioned table (some columns and all constraints of the sample table have been omitted in this example):

```
CREATE TABLE list_customers
   ( customer_id             NUMBER(6)
   , cust_first_name         VARCHAR2(20)
   , cust_last_name          VARCHAR2(20)
   , cust_address            CUST_ADDRESS_TYP
   , nls_territory           VARCHAR2(30)
   , cust_email              VARCHAR2(30))
   PARTITION BY LIST (nls_territory) (
```

```
   PARTITION asia VALUES ('CHINA', 'THAILAND'),
   PARTITION europe VALUES ('GERMANY', 'ITALY', 'SWITZERLAND'),
   PARTITION west VALUES ('AMERICA'),
   PARTITION east VALUES ('INDIA'),
   PARTITION rest VALUES (DEFAULT));
```

Partitioned Table with LOB Columns Example This statement creates a partitioned table `part_tab` with two partitions p1 and `p2`, and three LOB columns, `b`, `c`, and `d`. The statement uses the sample table `pm.print_media`, but the `LONG` column `press_release` is omitted because `LONG` columns are not supported in partitioning.

```
CREATE TABLE print_media_demo
   ( product_id NUMBER(6)
   , ad_id NUMBER(6)
   , ad_composite BLOB
   , ad_sourcetext CLOB
   , ad_finaltext CLOB
   , ad_fltextn NCLOB
   , ad_textdocs_ntab textdoc_tab
   , ad_photo BLOB
   , ad_graphic BFILE
   , ad_header adheader_typ
   ) NESTED TABLE ad_textdocs_ntab STORE AS textdocs_nestedtab_demo
      LOB (ad_composite, ad_photo, ad_finaltext)
      STORE AS(STORAGE (NEXT 20M))
   PARTITION BY RANGE (product_id)
      (PARTITION p1 VALUES LESS THAN (3000) TABLESPACE tbs_1
         LOB (ad_composite, ad_photo)
         STORE AS (TABLESPACE tbs_2 STORAGE (INITIAL 10M)),
       PARTITION P2 VALUES LESS THAN (MAXVALUE)
         LOB (ad_composite, ad_finaltext)
         STORE AS (TABLESPACE tbs_3)
       )
   TABLESPACE tbs_4;
```

Partition `p1` will be in tablespace `tbs_1`. The LOB data partitions for `ad_composite` and `ad_finaltext` will be in tablespace `tbs_2`. The LOB data partition for `ad_photo` will be in tablespace `tbs_1`. The storage attribute `INITIAL` is specified for LOB columns `ad_composite` and `ad_finaltext`. Other attributes will be inherited from the default table-level specification. The default LOB storage attributes not specified at the table level will be inherited from the tablespace `tbs_2` for columns `ad_composite` and `ad_finaltext` and tablespace tbs_1 for column `ad_photo`. LOB index partitions will be in the same tablespaces as the

corresponding LOB data partitions. Other storage attributes will be based on values of the corresponding attributes of the LOB data partitions and default attributes of the tablespace where the index partitions reside.

Partition p2 will be in the default tablespace tbs_4. The LOB data for ad_composite and ad_photo will be in tablespace tbs_3. The LOB data for ad_finaltext will be in tablespace tbs_4. The LOB index for columns ad_composite and ad_photo will be in tablespace tbs_3. The LOB index for column ad_finaltext will be in tablespace tbs_4.

Hash Partitioning Example The sample table oe.product_information is not partitioned. However, you might want to partition such a large table by hash for performance reasons, as shown in this example. (The tablespace names are hypothetical in this example.)

```
CREATE TABLE hash_products
    ( product_id          NUMBER(6)
    , product_name        VARCHAR2(50)
    , product_description VARCHAR2(2000)
    , category_id         NUMBER(2)
    , weight_class        NUMBER(1)
    , warranty_period     INTERVAL YEAR TO MONTH
    , supplier_id         NUMBER(6)
    , product_status      VARCHAR2(20)
    , list_price          NUMBER(8,2)
    , min_price           NUMBER(8,2)
    , catalog_url         VARCHAR2(50)
    , CONSTRAINT          product_status_lov
                          CHECK (product_status in ('orderable'
                                                    ,'planned'
                                                    ,'under development'
                                                    ,'obsolete')
 ) )
 PARTITION BY HASH (product_id)
 PARTITIONS 5
 STORE IN (tbs_1, tbs_2, tbs_3, tbs_4);
```

Composite-Partitioned Table Examples The table created in the "Range Partitioning Example" on page 15-73 divides data by time of sale. If you plan to access recent data according to distribution channel as well as time, then composite partitioning might be more appropriate. The following example creates a copy of that range_sales table, but with range-hash composite partitioning. The partitions with the most recent data are subpartitioned with both Oracle-defined

and user-defined subpartition names. (Constraints and storage attributes have been omitted from the example).

```
CREATE TABLE composite_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold         NUMBER(10,2)
    )
PARTITION BY RANGE (time_id)
SUBPARTITION BY HASH (channel_id)
  (PARTITION SALES_Q1_1998 VALUES LESS THAN (TO_DATE('01-APR-1998','DD-MON-YYYY')),
   PARTITION SALES_Q2_1998 VALUES LESS THAN (TO_DATE('01-JUL-1998','DD-MON-YYYY')),
   PARTITION SALES_Q3_1998 VALUES LESS THAN (TO_DATE('01-OCT-1998','DD-MON-YYYY')),
   PARTITION SALES_Q4_1998 VALUES LESS THAN (TO_DATE('01-JAN-1999','DD-MON-YYYY')),
   PARTITION SALES_Q1_1999 VALUES LESS THAN (TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION SALES_Q2_1999 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION SALES_Q3_1999 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION SALES_Q4_1999 VALUES LESS THAN (TO_DATE('01-JAN-2000','DD-MON-YYYY')),
   PARTITION SALES_Q1_2000 VALUES LESS THAN (TO_DATE('01-APR-2000','DD-MON-YYYY')),
   PARTITION SALES_Q2_2000 VALUES LESS THAN (TO_DATE('01-JUL-2000','DD-MON-YYYY'))
      SUBPARTITIONS 8,
   PARTITION SALES_Q3_2000 VALUES LESS THAN (TO_DATE('01-OCT-2000','DD-MON-YYYY'))
     (SUBPARTITION ch_c,
      SUBPARTITION ch_i,
      SUBPARTITION ch_p,
      SUBPARTITION ch_s,
      SUBPARTITION ch_t),
   PARTITION SALES_Q4_2000 VALUES LESS THAN (MAXVALUE)
      SUBPARTITIONS 4)
;
```

The following examples creates a partitioned table of customers based on the sample table `oe.customers`. In this example, the table is partitioned on the `credit_limit` column and list subpartitioned on the `nls_territory` column. The subpartition template determines the subpartitioning of any subsequently added partitions (unless you override the template by defining individual subpartitions). This composite partitioning makes it possible to query the table based on a credit limit range within a specified region:

```
CREATE TABLE customers_part (
   customer_id        NUMBER(6),
   cust_first_name    VARCHAR2(20),
```

```
   cust_last_name     VARCHAR2(20),
   nls_territory      VARCHAR2(30),
   credit_limit       NUMBER(9,2))
   PARTITION BY RANGE (credit_limit)
   SUBPARTITION BY LIST (nls_territory)
      SUBPARTITION TEMPLATE
         (SUBPARTITION east  VALUES
            ('CHINA', 'JAPAN', 'INDIA', 'THAILAND'),
          SUBPARTITION west VALUES
             ('AMERICA', 'GERMANY', 'ITALY', 'SWITZERLAND'),
          SUBPARTITION other VALUES (DEFAULT))
      (PARTITION p1 VALUES LESS THAN (1000),
       PARTITION p2 VALUES LESS THAN (2500),
       PARTITION p3 VALUES LESS THAN (MAXVALUE));
```

Object Column and Table Examples

Creating Object Tables: Examples Consider object type `department_typ`:

```
CREATE TYPE department_typ AS OBJECT
   ( d_name   VARCHAR2(100),
     d_address VARCHAR2(200) );
```

Object table `departments_obj_t` holds department objects of type `department_typ`:

```
CREATE TABLE departments_obj_t OF department_typ;
```

The following statement creates object table `salesreps` with a user-defined object type, `salesrep_typ`:

```
CREATE OR REPLACE TYPE salesrep_typ AS OBJECT
  ( repId NUMBER,
    repName VARCHAR2(64));

CREATE TABLE salesreps OF salesrep_typ;
```

Creating Tables with a Scoped REF: Example The following example uses the type `department_typ` and the table `departments_obj_t` (created in "Creating Object Tables: Examples" on page 15-77). A table with a scoped REF is then created.

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ SCOPE IS departments_obj_t );
```

The following statement creates a table with a REF column which has a referential integrity constraint defined on it:

```
CREATE TABLE employees_obj
   ( e_name   VARCHAR2(100),
     e_number NUMBER,
     e_dept   REF department_typ REFERENCES departments_obj_t);
```

Creating a Table with a User-Defined OID: Example This example creates an object type and a corresponding object table whose OID is primary key based:

```
CREATE TYPE employees_typ AS OBJECT
   (e_no NUMBER, e_address CHAR(30));

CREATE TABLE employees_obj_t OF employees_typ (e_no PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

You can subsequently reference the emp object table in either of the following two ways:

```
CREATE TABLE departments_t
   (d_no    NUMBER,
    mgr_ref REF employees_typ SCOPE IS employees_obj_t);

CREATE TABLE departments_t (
    d_no NUMBER,
    mgr_ref REF employees_typ
       CONSTRAINT mgr_in_emp REFERENCES employees_obj_t);
```

Specifying Constraints on Type Columns: Example

```
CREATE TYPE address_t AS OBJECT
  ( hno    NUMBER,
    street VARCHAR2(40),
    city   VARCHAR2(20),
    zip    VARCHAR2(5),
    phone  VARCHAR2(10) );

CREATE TYPE person AS OBJECT
  ( name        VARCHAR2(40),
    dateofbirth DATE,
    homeaddress address,
    manager     REF person );

CREATE TABLE persons OF person
```

```
( homeaddress NOT NULL,
    UNIQUE (homeaddress.phone),
    CHECK (homeaddress.zip IS NOT NULL),
    CHECK (homeaddress.city <> 'San Francisco') );
```

CREATE TABLESPACE

Purpose

Use the CREATE TABLESPACE statement to create a **tablespace**, which is an allocation of space in the database that can contain persistent schema objects.

When you create a tablespace, it is initially a read / write tablespace. You can subsequently use the ALTER TABLESPACE statement to take the tablespace offline or online, add datafiles to it, or make it a read-only tablespace.

You can also drop a tablespace from the database with the DROP TABLESPACE statement.

You can use the CREATE TEMPORARY TABLESPACE statement to create tablespaces that contain schema objects only for the duration of a session.

See Also:

- *Oracle9i Database Concepts* for information on tablespaces
- ALTER TABLESPACE on page 11-101 for information on modifying tablespaces
- DROP TABLESPACE on page 17-10 for information on dropping tablespaces
- CREATE TEMPORARY TABLESPACE on page 15-92

Prerequisites

You must have CREATE TABLESPACE system privilege.

Before you can create a tablespace, you must create a database to contain it, and the database must be open.

See Also: CREATE DATABASE on page 13-22

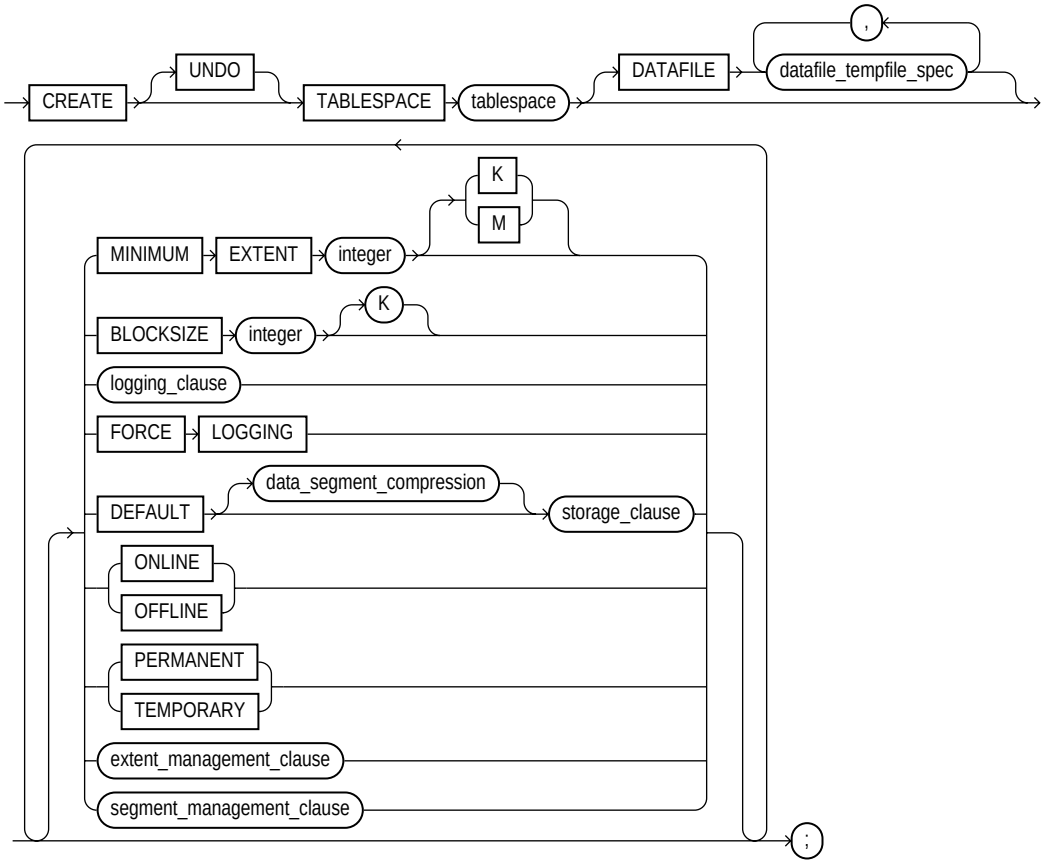
To use objects in a tablespace other than the SYSTEM tablespace:

- If you are running the database in rollback undo mode, at least one rollback segment (other than the SYSTEM rollback segment) must be online.
- If you are running the database in Automatic Undo Management mode, at least one UNDO tablespace must be online.

See Also: CREATE ROLLBACK SEGMENT on page 14-80

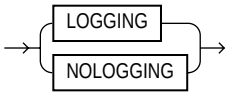
Syntax

create_tablespace::=

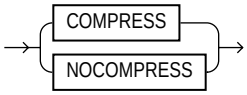


(*datafile_tempfile_spec::=* on page 7-39—part of *file_specification*, *logging_clause::=* on page 15-82, *data_segment_compression::=* on page 15-82, *storage_clause::=* on page 7-58, *extent_management_clause::=* on page 15-82, *segment_management_clause::=* on page 15-82)

logging_clause::=

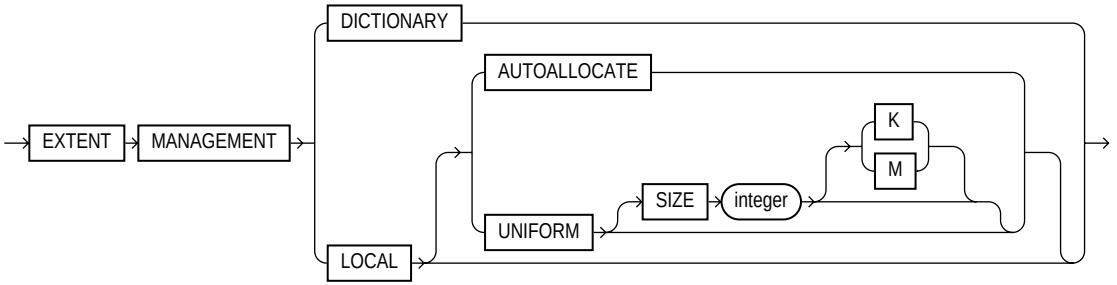


data_segment_compression::=



(*storage_clause* on page 7-56)

extent_management_clause::=



segment_management_clause::=

Keywords and Parameters

UNDO

Specify UNDO to create an undo tablespace. When you run the database in Automatic Undo Management mode, Oracle manages undo space using the undo tablespace instead of rollback segments. This clause is useful if you are now running in Automatic Undo Management mode but your database was not created in Automatic Undo Management mode.

Oracle always assigns an undo tablespace when you start up the database in Automatic Undo Management mode. If no undo tablespace has been assigned to

this instance, then Oracle will use the SYSTEM rollback segment. You can avoid this by creating an undo tablespace, which Oracle will implicitly assign to the instance if no other undo tablespace is currently assigned.

Restrictions on undo tablespaces:

- You cannot create database objects in this tablespace. It is reserved for system-managed undo data.
- The only clauses you can specify for an undo tablespace are the DATAFILE clause and the *extent_management_clause* to specify local extent management. (You cannot specify dictionary extent management using the *extent_management_clause*.) All undo tablespaces are created permanent, read / write, and in logging mode. Values for MINIMUM EXTENT and DEFAULT STORAGE are system generated.

  See Also:

  - *Oracle9i Database Administrator's Guide* for information on Automatic Undo Management and undo tablespaces
  - CREATE DATABASE on page 13-22 for information on creating an undo tablespace implicitly or explicitly during database creation
  - ALTER TABLESPACE for information about altering undo tablespaces
  - DROP TABLESPACE for information about dropping undo tablespaces
  - *Oracle9i Database Reference* for information on opening a database instance in Automatic Undo Management mode using the UNDO_MANAGEMENT parameter
  - "Creating an Undo Tablespace: Example" on page 15-89

tablespace

Specify the name of the tablespace to be created.

DATAFILE *datafile_tempfile_spec*

Specify the datafile or files to make up the tablespace.

Note: For operating systems that support raw devices, the REUSE keyword of *datafile_tempfile_spec* has no meaning when specifying a raw device as a datafile. Such a CREATE TABLESPACE statement will succeed whether or not you specify REUSE.

The DATAFILE clause is optional only if the DB_CREATE_FILE_DEST initialization parameter is set. In this case, Oracle creates a system-named 100MB file in the default file destination specified in the parameter. The file has AUTOEXTEND enabled and an unlimited maximum size.

See Also:

- *file_specification* on page 7-39 for a full description, including the AUTOEXTEND parameter
- "Enabling Autoextend for a Tablespace: Example" on page 15-90 and "Creating Oracle-managed Files: Examples" on page 15-91

MINIMUM EXTENT Clause

Specify the minimum size of an extent in the tablespace. This clause lets you control free space fragmentation in the tablespace by ensuring that every used or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*.

Note: This clause is not relevant for a dictionary-managed temporary tablespace.

See Also: *Oracle9i Database Concepts* for more information about using MINIMUM EXTENT to control fragmentation and "Specifying Minimum Extent Size: Example" on page 15-90

BLOCKSIZE Clause

Use the BLOCKSIZE clause to specify a nonstandard block size for the tablespace. In order to specify this clause, you must have the DB_CACHE_SIZE and at least one DB_*n*K_CACHE_SIZE parameter set, and the integer you specify in this clause must correspond with the setting of one DB_*n*K_CACHE_SIZE parameter setting.

Restriction on BLOCKSIZE: You cannot specify nonstandard block sizes for a temporary tablespace (that is, if you also specify TEMPORARY) or if you intend to assign this tablespace as the temporary tablespace for any users.

> **See Also:** *Oracle9i Database Administrator's Guide* for information on allowing multiple block sizes in the buffer cache, and for restrictions on using multiple block sizes in partitioned objects

logging_clause

Specify the default logging attributes of all tables, indexes, materialized views, materialized view logs, and partitions within the tablespace. LOGGING is the default.

The tablespace-level logging attribute can be overridden by logging specifications at the table, index, materialized view, materialized view log, and partition levels.

> **See Also:** *logging_clause* on page 7-45 for a full description of this clause

FORCE LOGGING

Use this clause to put the tablespace into FORCE LOGGING mode. Oracle will log all changes to all objects in the tablespace except changes to temporary segments, overriding any NOLOGGING setting for individual objects. The database must be open and in READ WRITE mode.

This setting does not exclude the NOLOGGING attribute. That is, you can specify both FORCE LOGGING and NOLOGGING. In this case, NOLOGGING is the default logging mode for objects subsequently created in the tablespace, but Oracle ignores this default as long as the tablespace (or the database) is in FORCE LOGGING mode. If you subsequently take the tablespace out of FORCE LOGGING mode, then the NOLOGGING default is once again enforced.

> **Note:** FORCE LOGGING mode can have performance effects. Please refer to *Oracle9i Database Administrator's Guide* for information on when to use this setting.

Restriction on forced logging: You cannot specify FORCE LOGGING for an undo or temporary tablespace.

DEFAULT *storage_clause*

Specify the default storage parameters for all objects created in the tablespace.

For a dictionary-managed temporary tablespace, Oracle considers only the NEXT parameter of the *storage_clause*.

> **See Also:** *storage_clause* on page 7-56 for information on storage parameters and "Creating a Tablespace with Default Storage: Example" on page 15-90

ONLINE | OFFLINE Clauses

ONLINE Specify ONLINE to make the tablespace available immediately after creation to users who have been granted access to the tablespace. This is the default.

OFFLINE Specify OFFLINE to make the tablespace unavailable immediately after creation.

The data dictionary view DBA_TABLESPACES indicates whether each tablespace is online or offline.

PERMANENT | TEMPORARY Clauses

PERMANENT Specify PERMANENT if the tablespace will be used to hold permanent objects. This is the default.

TEMPORARY Specify TEMPORARY if the tablespace will be used only to hold temporary objects, for example, segments used by implicit sorts to handle ORDER BY clauses.

Temporary tablespaces created with this clause are always dictionary managed, so you cannot specify the EXTENT MANAGEMENT LOCAL clause. To create a locally managed temporary tablespace, use the CREATE TEMPORARY TABLESPACE statement.

Note: Oracle Corporation strongly recommends that you create locally managed temporary tablespaces containing tempfiles by using the CREATE TEMPORARY TABLESPACE statement. The creation of new dictionary-managed tablespaces is scheduled for desupport.

Restriction on temporary tablespaces: If you specify TEMPORARY, then you cannot specify the BLOCKSIZE clause.

extent_management_clause

The *extent_management_clause* lets you specify how the extents of the tablespace will be managed.

> **Note:** Once you have specified extent management with this clause, you can change extent management only by migrating the tablespace.

- Specify LOCAL if you want the tablespace to be locally managed. Locally managed tablespaces have some part of the tablespace set aside for a bitmap. This is the default.
  - AUTOALLOCATE specifies that the tablespace is system managed. Users cannot specify an extent size. This is the default if the COMPATIBLE initialization parameter is set to 9.0.0 or higher.
  - UNIFORM specifies that the tablespace is managed with uniform extents of SIZE bytes. Use *K* or *M* to specify the extent size in kilobytes or megabytes. The default SIZE is 1 megabyte.
- Specify DICTIONARY if you want the tablespace to be managed using dictionary tables. This is the default if the COMPATIBLE initialization parameter is set less than 9.0.0.

  Restriction: You cannot specify DICTIONARY if the SYSTEM tablespace of the database is locally managed.

  > **Note:** Oracle Corporation strongly recommends that you create only locally managed tablespaces. Locally managed tablespaces are much more efficiently managed than dictionary-managed tablespaces. The creation of new dictionary-managed tablespaces is scheduled for desupport.

If you do not specify the *extent_management_clause*, then Oracle interprets the COMPATIBLE setting, the MINIMUM EXTENT clause and the DEFAULT *storage_clause* to determine extent management. If the COMPATIBLE initialization

parameter is less than 9.0.0, then Oracle creates a dictionary managed tablespace. If COMPATIBLE = 9.0.0 or higher:

- If you do not specify the DEFAULT *storage_clause* at all, then Oracle creates a locally managed autoallocated tablespace.
- If you did specify the DEFAULT *storage_clause*:
  - If you specified the MINIMUM EXTENT clause, then Oracle evaluates whether the values of MINIMUM EXTENT, INITIAL, and NEXT are equal and the value of PCTINCREASE is 0. If so, Oracle creates a locally managed uniform tablespace with extent size = INITIAL. If the MINIMUM EXTENT, INITIAL, and NEXT parameters are not equal, or if PCTINCREASE is not 0, Oracle ignores any extent storage parameters you may specify and creates a locally managed, autoallocated tablespace.
  - If you did not specify MINIMUM EXTENT clause, then Oracle evaluates only whether the storage values of INITIAL and NEXT are equal and PCTINCREASE is 0. If so, the tablespace is locally managed and uniform. Otherwise, the tablespace is locally managed and autoallocated.

    See Also: *Oracle9i Database Concepts* for a discussion of locally managed tablespaces

Restrictions on extent management:

- A permanent locally managed tablespace can contain only permanent objects. If you need a locally managed tablespace to store temporary objects (for example, if you will assign it as a user's temporary tablespace, use the CREATE TEMPORARY TABLESPACE statement.
- If you specify LOCAL, then you cannot specify DEFAULT *storage_clause*, MINIMUM EXTENT, or TEMPORARY.

  See Also: *Oracle9i Database Migration* for information on changing extent management by migrating tablespaces and "Creating a Locally Managed Tablespace: Example" on page 15-90

segment_management_clause

The *segment_management_clause* is relevant only for permanent, locally managed tablespaces. It lets you specify whether Oracle should track the used and free space in the segments in the tablespace using free lists or bitmaps.

MANUAL Specify MANUAL if you want Oracle to manage the free space of segments in the tablespace using free lists.

AUTO Specify AUTO if you want Oracle to manage the free space of segments in the tablespace using a bitmap. If you specify AUTO, then Oracle ignores any specification for PCTUSED, FREELIST, and FREELIST GROUPS in subsequent storage specifications for objects in this tablespace. This setting is called **automatic segment-space management**.

To determine the segment management of an existing tablespace, query the SEGMENT_SPACE_MANAGEMENT column of the DBA_TABLESPACES or USER_TABLESPACES data dictionary view.

Notes: If you specify AUTO, then:

- If you set extent management to LOCAL UNIFORM, then you must ensure that each extent contains at least 5 database blocks, given the database block size.
- If you set extent management to LOCAL AUTOALLOCATE, and if the database block size is 16K or greater, then Oracle manages segment space by creating extents with a minimum size of 1M.

Restrictions on the AUTO clause:

- You can specify this clause only for permanent, locally managed tablespace.
- You cannot specify this clause for the SYSTEM tablespace.

See Also:

- *Oracle9i Database Administrator's Guide* for information on automatic segment-space management and when to use it
- *Oracle9i Database Reference* for information on the data dictionary views
- "Specifying Segment Space Management for a Tablespace: Example" on page 15-90

Examples

Creating an Undo Tablespace: Example The following example creates a 10 MB undo tablespace undots1 with datafile undotbs_1a.f:

```
CREATE UNDO TABLESPACE undots1
   DATAFILE 'undotbs_1a.f'
   SIZE 10M AUTOEXTEND ON;
```

Creating a Tablespace with Default Storage: Example This statement creates a tablespace named `tbs_01` with one datafile:

```
CREATE TABLESPACE tbs_01
   DATAFILE 'tbs_f2.dat' SIZE 20M
   DEFAULT STORAGE (INITIAL 128K NEXT 128K
                    MINEXTENTS 1 MAXEXTENTS 999)
   ONLINE;
```

Enabling Autoextend for a Tablespace: Example This statement creates a tablespace named `tbs_02` with one datafile. When more space is required, 500 kilobyte extents will be added up to a maximum size of 10 megabytes:

```
CREATE TABLESPACE tbs_02
   DATAFILE 'diskb:tbs_f5.dat' SIZE 500K REUSE
   AUTOEXTEND ON NEXT 500K MAXSIZE 100M;
```

Specifying Minimum Extent Size: Example This statement creates tablespace `tbs_03` with one datafile and allocates every extent as a multiple of 500K:

```
CREATE TABLESPACE tbs_03
   DATAFILE 'tbs_f03.dbf' SIZE 20M
   MINIMUM EXTENT 500K
   DEFAULT STORAGE (INITIAL 128K NEXT 128K)
   LOGGING;
```

Creating a Locally Managed Tablespace: Example In the following statement, we assume that the database block size is 2K.

```
CREATE TABLESPACE tbs_04 DATAFILE 'file_1.f' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128K;
```

This statement creates a locally managed tablespace in which every extent is 128K and each bit in the bit map describes 64 blocks.

Specifying Segment Space Management for a Tablespace: Example The following example creates a tablespace with automatic segment-space management:

```
CREATE TABLESPACE auto_seg_ts DATAFILE 'file_2.f' SIZE 1M
   EXTENT MANAGEMENT LOCAL
   SEGMENT SPACE MANAGEMENT AUTO;
```

Creating Oracle-managed Files: Examples The following example sets the default location for datafile creation and creates a tablespace with a datafile in the default location. The datafile is 100M and is autoextensible with an unlimited maximum size:

```
ALTER SYSTEM SET DB_CREATE_FILE_DEST = '$ORACLE_HOME/rdbms/log';

CREATE TABLESPACE omf_ts1;
```

The following example creates a tablespace with an Oracle managed datafile of 100M that is not autoextensible:

```
CREATE TABLESPACE omf_ts2 DATAFILE AUTOEXTEND OFF;
```

CREATE TEMPORARY TABLESPACE

Purpose

Use the CREATE TEMPORARY TABLESPACE statement to create a locally managed **temporary tablespace**, which is an allocation of space in the database that can contain schema objects for the duration of a session. If you subsequently assign this temporary tablespace to a particular user, then Oracle will also use this tablespace for sorting operations in transactions initiated by that user.

To create a tablespace to contain persistent schema objects, use the CREATE TABLESPACE statement.

To create a temporary tablespace that is dictionary managed, use the CREATE TABLESPACE statement with the TEMPORARY clause.

Note: Media recovery does not recognize tempfiles.

See Also:

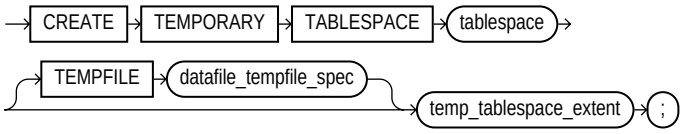
- CREATE TABLESPACE on page 15-80 for information on creating tablespaces to store persistent schema objects and dictionary-managed temporary tablespaces
- CREATE USER on page 16-32 for information on assigning a temporary tablespace to a user

Prerequisites

You must have the CREATE TABLESPACE system privilege.

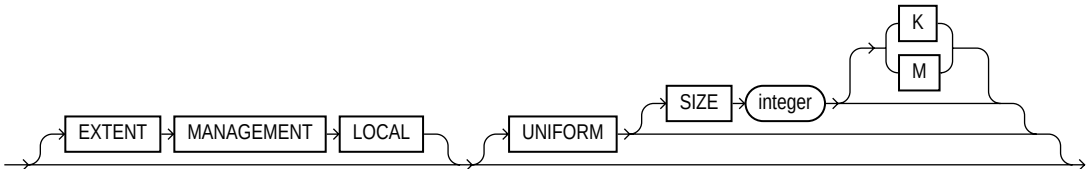
Syntax

create_temporary_tablespace::=



(*datafile_tempfile_spec::=* on page 7-39, *temp_tablespace_extent::=* on page 15-93)

temp_tablespace_extent::=



Keywords and Parameters

tablespace

Specify the name of the temporary tablespace.

TEMPFILE *datafile_tempfile_spec*

Specify the tempfiles that make up the tablespace.

You can omit the `TEMPFILE` clause only if the `DB_CREATE_FILE_DEST` initialization parameter has been set. In this case, Oracle creates a 100 MB Oracle-managed tempfile in the default file destination specified in the parameter. The file has `AUTOEXTEND` enabled and an unlimited maximum size. If the `DB_CREATE_FILE_DEST` parameter is not set, then you must specify the `TEMPFILE` clause.

Note: On some operating systems, Oracle does not allocate space for the tempfile until the tempfile blocks are actually accessed. This delay in space allocation results in faster creation and resizing of tempfiles, but it requires that sufficient disk space is available when the tempfiles are later used. Please refer to the Oracle documentation for your operating system to determine whether Oracle allocates tempfile space in this way on your system.

See Also: *file_specification* on page 7-39 for a full description, including the `AUTOEXTEND` parameter

temp_tablespace_extent

The *temp_tablespace_extent* clause lets you specify how the tablespace is managed.

EXTENT MANAGEMENT LOCAL This clause indicates that some part of the tablespace is set aside for a bitmap. All temporary tablespaces created with the CREATE TEMPORARY TABLESPACE statement have locally managed extents, so this clause is optional. To create a dictionary-managed temporary tablespace, use the CREATE TABLESPACE statement with the TEMPORARY clause.

UNIFORM All extents of temporary tablespaces are the same size (uniform), so this keyword is optional. However, you must specify UNIFORM in order to specify SIZE.

SIZE *integer* Specify in bytes the size of the tablespace extents. Use K or M to specify the size in kilobytes or megabytes.

If you do not specify SIZE, then Oracle uses the default extent size of 1M.

> **See Also:** *Oracle9i Database Concepts* for a discussion of locally managed tablespaces

Example

Creating a Temporary Tablespace: Example This statement shows how the temporary tablespace that serves as the default temporary tablespace for database users in the sample database was created:

```
CREATE TEMPORARY TABLESPACE temp_demo
   TEMPFILE 'temp01.dbf' SIZE 5M AUTOEXTEND ON;
```

If we assume the default database block size of 2K, and that each bit in the map represents one extent, then each bit maps 2,500 blocks.

The following example sets the default location for datafile creation and then creates a tablespace with an Oracle-managed tempfile in the default location. The tempfile is 100 M and is autoextensible with unlimited maximum size (the default values for Oracle-managed files):

```
ALTER SYSTEM SET DB_CREATE_FILE_DEST = '$ORACLE_HOME/rdbms/log';

CREATE TEMPORARY TABLESPACE tbs_05;
```

CREATE TRIGGER

Purpose

Use the CREATE TRIGGER statement to create and enable a **database trigger**, which is

- A stored PL/SQL block associated with a table, a schema, or the database or
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

Oracle automatically executes a trigger when specified conditions occur.

When you create a trigger, Oracle enables it automatically. You can subsequently disable and enable a trigger with the DISABLE and ENABLE clause of the ALTER TRIGGER or ALTER TABLE statement.

See Also:

- *Oracle9i Database Concepts* for a description of the various types of triggers
- *Oracle9i Application Developer's Guide - Fundamentals* for more information on how to design triggers
- ALTER TRIGGER on page 12-2 and ALTER TABLE on page 11-2 for information on enabling, disabling, and compiling triggers
- DROP TRIGGER on page 17-13 for information on dropping a trigger

Prerequisites

Before a trigger can be created, the user SYS must run a SQL script commonly called DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

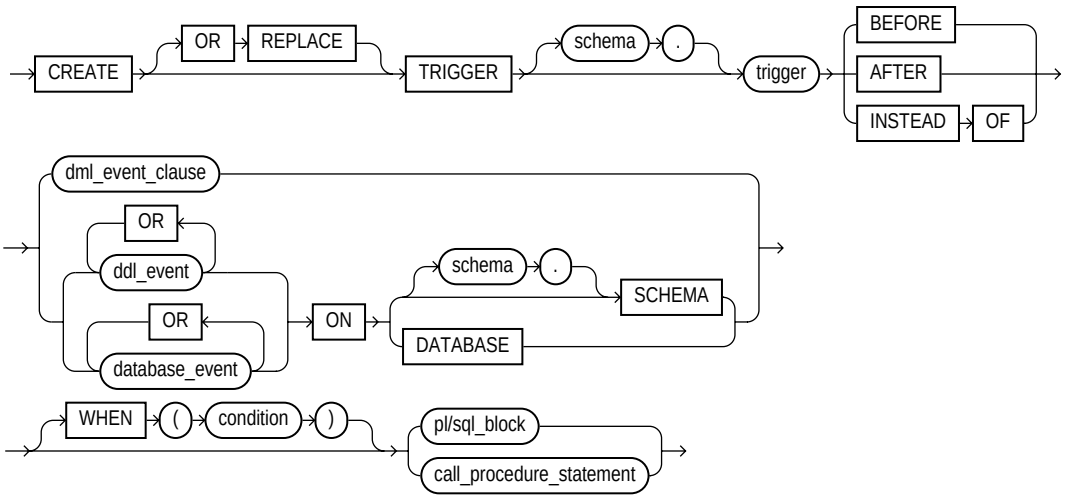
- To create a trigger in your own schema on a table in your own schema or on your own schema (SCHEMA), you must have the CREATE TRIGGER privilege.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*schema*.SCHEMA), you must have the CREATE ANY TRIGGER privilege.

- In addition to the preceding privileges, to create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

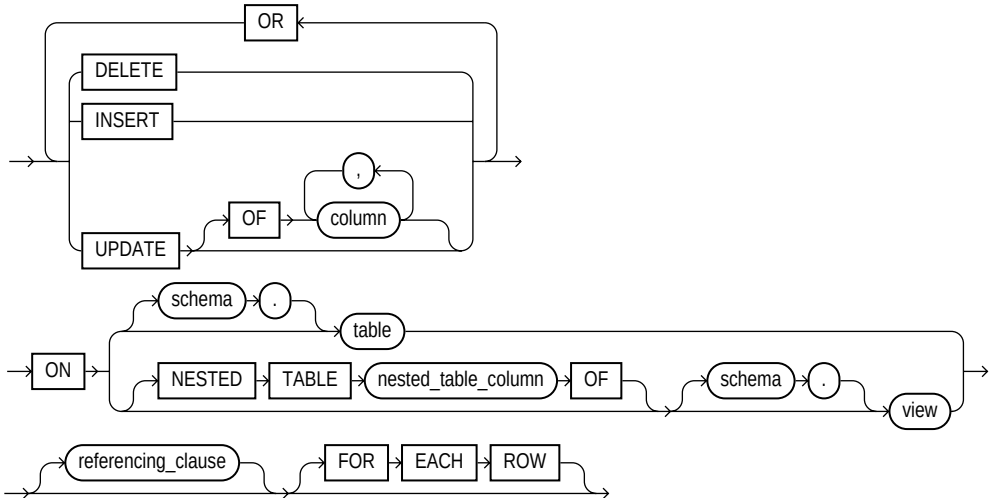
If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.

Syntax

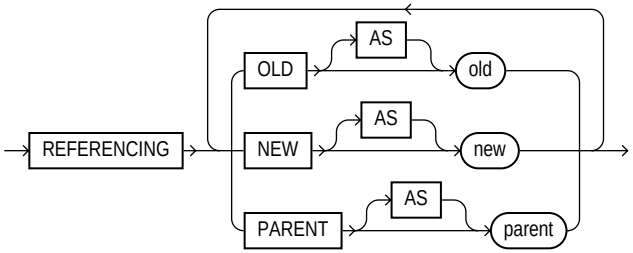
create_trigger::=



dml_event_clause::=



referencing_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the trigger if it already exists. Use this clause to change the definition of an existing trigger without first dropping it.

schema

Specify the schema to contain the trigger. If you omit *schema*, then Oracle creates the trigger in your own schema.

trigger

Specify the name of the trigger to be created.

If a trigger produces compilation errors, then it is still created, but it fails on execution. This means it effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

Note: If you create a trigger on a base table of a materialized view, then you must ensure that the trigger does not fire during a refresh of the materialized view. (During refresh, the DBMS_MVIEW procedure I_AM_A_REFRESH returns TRUE.)

BEFORE

Specify BEFORE to cause Oracle to fire the trigger before executing the triggering event. For row triggers, the trigger is fired before each affected row is changed.

Restrictions on BEFORE triggers:

- You cannot specify a BEFORE trigger on a view or an object view.
- You can write to the :NEW value but not to the :OLD value.

AFTER

Specify AFTER to cause Oracle to fire the trigger after executing the triggering event. For row triggers, the trigger is fired after each affected row is changed.

Restrictions on AFTER triggers:

- You cannot specify an AFTER trigger on a view or an object view.
- You cannot write either the :OLD or the :NEW value.

Note: When you create a materialized view log for a table, Oracle implicitly creates an AFTER ROW trigger on the table. This trigger inserts a row into the materialized view log whenever an INSERT, UPDATE, or DELETE statement modifies the table's data. You cannot control the order in which multiple row triggers fire. Therefore, you should not write triggers intended to affect the content of the materialized view.

See Also: CREATE MATERIALIZED VIEW LOG on page 14-32 for more information on materialized view logs

INSTEAD OF

Specify INSTEAD OF to cause Oracle to fire the trigger instead of executing the triggering event. INSTEAD OF triggers are valid for DML events on views. They are not valid for DDL or database events.

If a view is inherently updatable and has INSTEAD OF triggers, then the triggers take preference. In other words, Oracle fires the triggers instead of performing DML on the view.

If the view belongs to a hierarchy, then the trigger is not inherited by subviews.

Restrictions on INSTEAD OF triggers:

- INSTEAD OF triggers are valid only for views. You cannot specify an INSTEAD OF trigger on a table.
- You can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW value.

Note: You can create multiple triggers of the same type (BEFORE, AFTER, or INSTEAD OF) that fire for the same statement on the same table. The order in which Oracle fires these triggers is indeterminate. If your application requires that one trigger be fired before another of the same type for the same statement, combine these triggers into a single trigger whose trigger action performs the trigger actions of the original triggers in the appropriate order.

See Also: "Creating an INSTEAD OF Trigger: Example" on page 15-108

dml_event_clause

The *dml_event_clause* lets you specify one of three DML statements that can cause the trigger to fire. Oracle fires the trigger in the existing user transaction.

See Also: "Creating a DML Trigger: Examples" on page 15-106

DELETE

Specify DELETE if you want Oracle to fire the trigger whenever a DELETE statement removes a row from the table or removes an element from a nested table.

INSERT

Specify INSERT if you want Oracle to fire the trigger whenever an INSERT statement adds a row to table or adds an element to a nested table.

UPDATE

Specify UPDATE if you want Oracle to fire the trigger whenever an UPDATE statement changes a value in one of the columns specified after OF. If you omit OF, then Oracle fires the trigger whenever an UPDATE statement changes a value in any column of the table or nested table.

For an UPDATE trigger, you can specify object type, varray, and REF columns after OF to indicate that the trigger should be fired whenever an UPDATE statement changes a value in one of the columns. However, you cannot change the values of these columns in the body of the trigger itself.

Note: Using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns does not cause Oracle to fire triggers defined on the table containing the columns or the attributes.

Restrictions on triggers on UPDATE operations:

- You cannot specify UPDATE OF for an INSTEAD OF trigger. Oracle fires INSTEAD OF triggers whenever an UPDATE changes a value in any column of the view.
- You cannot specify a nested table or LOB column in the UPDATE OF clause.

> **See Also:** AS *subquery* of CREATE VIEW on page 16-39 for a list of constructs that prevent inserts, updates, or deletes on a view

Performing DML operations directly on nested table columns does not cause Oracle to fire triggers defined on the table containing the nested table column.

ddl_event

Specify one or more types of DDL statements that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can create BEFORE and AFTER triggers for these events. Oracle fires the trigger in the existing user transaction.

Restriction on triggers on DDL events: You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

See Also: "Creating a DDL Trigger: Example" on page 15-107

The following *ddl_event* values are valid:

ALTER Specify ALTER to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary.

Restriction on triggers on ALTER operations: The trigger will not be fired by an ALTER DATABASE statement.

ANALYZE Specify ANALYZE to fire the trigger whenever Oracle collects or deletes statistics or validates the structure of a database object.

ASSOCIATE STATISTICS Specify ASSOCIATE STATISTICS to fire the trigger whenever Oracle associates a statistics type with a database object.

AUDIT Specify AUDIT to fire the trigger whenever Oracle tracks the occurrence of a SQL statement or tracks operations on a schema object.

COMMENT Specify COMMENT to fire the trigger whenever a comment on a database object is added to the data dictionary.

CREATE Specify CREATE to fire the trigger whenever a CREATE statement adds a new database object to the data dictionary.

Restriction on triggers on CREATE operations: The trigger will not be fired by a CREATE DATABASE or CREATE CONTROLFILE statement.

DISASSOCIATE STATISTICS Specify DISASSOCIATE STATISTICS to fire the trigger whenever Oracle disassociates a statistics type from a database object.

DROP Specify DROP to fire the trigger whenever a DROP statement removes a database object from the data dictionary.

GRANT Specify GRANT to fire the trigger whenever a user grants system privileges or roles or object privileges to another user or to a role.

NOAUDIT Specify NOAUDIT to fire the trigger whenever a NOAUDIT statement instructs Oracle to stop tracking a SQL statement or operations on a schema object.

RENAME Specify RENAME to fire the trigger whenever a RENAME statement changes the name of a database object.

REVOKE Specify REVOKE to fire the trigger whenever a REVOKE statement removes system privileges or roles or object privileges from a user or role.

TRUNCATE Specify TRUNCATE to fire the trigger whenever a TRUNCATE statement removes the rows from a table or cluster and resets its storage characteristics.

DDL Specify DDL to fire the trigger whenever any of the preceding DDL statements is issued.

database_event

Specify one or more particular states of the database that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. For each of these triggering events, Oracle opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction).

> **See Also:** "Creating a Database Event Trigger: Example" on page 15-108

SERVERERROR Specify SERVERERROR to fire the trigger whenever a server error message is logged.

The following errors do not cause a SERVERERROR trigger to fire:

- ORA-01403: data not found
- ORA-01422: exact fetch returns more than requested number of rows
- ORA-01423: error encountered while checking for extra rows in exact fetch
- ORA-01034: ORACLE not available
- ORA-04030: out of process memory

LOGON Specify LOGON to fire the trigger whenever a client application logs onto the database.

LOGOFF Specify LOGOFF to fire the trigger whenever a client applications logs off the database.

STARTUP Specify STARTUP to fire the trigger whenever the database is opened.

SHUTDOWN Specify SHUTDOWN to fire the trigger whenever an instance of the database is shut down.

SUSPEND Specify SUSPEND to fire the trigger whenever a server error causes a transaction to be suspended.

Notes:

- Only AFTER triggers are relevant for LOGON, STARTUP, SERVERERROR, and SUSPEND.
- Only BEFORE triggers are relevant for LOGOFF and SHUTDOWN.
- AFTER STARTUP and BEFORE SHUTDOWN triggers apply only to DATABASE.

See Also: *PL/SQL User's Guide and Reference* for more information on autonomous transaction scope

ON *table* | *view*

The ON clause lets you determine the database object on which the trigger is to be created.

table | *view*

Specify the *schema* and *table* or *view* name of one of the following on which the trigger is to be created:

- Table or view
- Object table or object view
- A column of nested-table type

If you omit *schema*, then Oracle assumes the table is in your own schema. You can create triggers on index-organized tables.

Restriction on *schema*: You cannot create a trigger on a table in the schema SYS.

NESTED TABLE Clause

Specify the *nested_table_column* of a view upon which the trigger is being defined. Such a trigger will fire only if the DML operates on the elements of the nested table.

Restriction on triggers on nested tables: You can specify NESTED TABLE only for INSTEAD OF triggers.

DATABASE

Specify DATABASE to define the trigger on the entire database. The trigger fires whenever any database user initiates the triggering event.

SCHEMA

Specify SCHEMA to define the trigger on the current schema. The trigger fires whenever any user connected as *schema* initiates the triggering event.

See Also: "Creating a SCHEMA Trigger: Example" on page 15-109

referencing_clause

The *referencing_clause* lets you specify correlation names. You can use correlation names in the PL/SQL block and WHEN condition of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, use this clause to specify different correlation names to avoid confusion between the table name and the correlation name.

- If the trigger is defined on a nested table, then OLD and NEW refer to the row of the nested table, and PARENT refers to the current row of the parent table.
- If the trigger is defined on an object table or view, then OLD and NEW refer to object instances.

Restriction on the *referencing_clause*: The *referencing_clause* is not valid with INSTEAD OF triggers on CREATE DDL events.

FOR EACH ROW

Specify FOR EACH ROW to designate the trigger as a row trigger. Oracle fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the WHEN condition.

Except for INSTEAD OF triggers, if you omit this clause, then the trigger is a statement trigger. Oracle fires a statement trigger only once when the triggering statement is issued if the optional trigger constraint is met.

INSTEAD OF trigger statements are implicitly activated for each row.

Restriction on row triggers: This clause is valid only for DML event triggers (not DDL or database event triggers).

WHEN Clause

Specify the trigger restriction, which is a SQL condition that must be satisfied for Oracle to fire the trigger. See the syntax description of *condition* in Chapter 5, "Conditions". This condition must contain correlation names and cannot contain a query.

The NEW and OLD keywords, when specified in the WHEN clause, are not considered bind variables, so are not preceded by a colon (:). However, you must precede NEW and OLD with a colon in all references other than the WHEN clause.

> **See Also:** "Calling a Procedure in a Trigger Body: Example" on page 15-107

Restrictions on trigger conditions:

- If you specify this clause for a DML event trigger, then you must also specify FOR EACH ROW. Oracle evaluates this condition for each row affected by the triggering statement.
- You cannot specify trigger restrictions for INSTEAD OF trigger statements.
- You can reference object columns or their attributes, or varray, nested table, or LOB columns. You cannot invoke PL/SQL functions or methods in the trigger restriction.

pl/sql_block

Specify the PL/SQL block that Oracle executes to fire the trigger.

The PL/SQL block of a database trigger can contain one of a series of built-in functions in the SYS schema designed solely to extract system event attributes. These functions can be used *only* in the PL/SQL block of a database trigger.

Restrictions on trigger implementation:

- The PL/SQL block of a trigger cannot contain transaction control SQL statements (`COMMIT`, `ROLLBACK`, `SAVEPOINT`, and `SET CONSTRAINT`) if the block is executed within the same transaction.
- You can reference and use LOB columns in the trigger action inside the PL/SQL block. You can modify the `:NEW` values but not the `:OLD` values of LOB columns within the trigger action.

See Also:

- *PL/SQL User's Guide and Reference* for information on PL/SQL, including how to write PL/SQL blocks
- *Oracle9i Application Developer's Guide - Fundamentals* for information on these functions
- "Calling a Procedure in a Trigger Body: Example" on page 15-107

call_procedure_statement

The *`call_procedure_statement`* lets you call a stored procedure, rather than specifying the trigger code inline as a PL/SQL block. The syntax of this statement is the same as that for `CALL` on page 12-66, with the following exceptions:

- You cannot specify the `INTO` clause of `CALL`, because it applies only to functions.
- You cannot specify bind variables in *`expr`*.
- To reference columns of tables on which the trigger is being defined, you must specify `:NEW` and `:OLD`.

See Also: "Calling a Procedure in a Trigger Body: Example" on page 15-107

Examples

Creating a DML Trigger: Examples This example creates a `BEFORE` statement trigger named `emp_permit_changes` in the schema `hr`. You would write such a trigger to place restrictions on DML statements issued on this table (such as when such statements could be issued).

```
CREATE TRIGGER hr.emp_permit_changes
```

```
BEFORE
DELETE OR INSERT OR UPDATE
ON hr.employees
   < pl/sql block >
```

Oracle fires this trigger whenever a DELETE, INSERT, or UPDATE statement affects the employees table in the schema hr. The trigger emp_permit_changes is a BEFORE statement trigger, so Oracle fires it once before executing the triggering statement.

This example creates a BEFORE row trigger named salary_check in the schema hr. The PL/SQL block might specify, for example, that the employee's salary must fall within the established salary range for the employee's job:

```
CREATE TRIGGER hr.salary_check
      BEFORE INSERT OR UPDATE OF salary, job_id ON hr.employees
      FOR EACH ROW
         WHEN (new.job_id <> 'AD_VP')
      < pl/sql_block >
```

Oracle fires this trigger whenever one of the following statements is issued:

- An INSERT statement that adds rows to the employees table
- An UPDATE statement that changes values of the salary or job_id columns of the employees table

salary_check is a BEFORE row trigger, so Oracle fires it before changing each row that is updated by the UPDATE statement or before adding each row that is inserted by the INSERT statement.

salary_check has a trigger restriction that prevents it from checking the salary of the administrative vice president (AD_VP).

Creating a DDL Trigger: Example This example creates an AFTER statement trigger on any DDL statement CREATE. Such a trigger can be used to audit the creation of new data dictionary objects in your schema.

```
CREATE TRIGGER audit_db_object AFTER CREATE
   ON SCHEMA
      < pl/sql_block >
```

Calling a Procedure in a Trigger Body: Example You could create the salary_check trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume you have defined a

procedure hr.salary_check, which verifies that an employee's salary is in an appropriate range. Then you could create the trigger salary_check as follows:

```
CREATE TRIGGER hr.salary_check
   BEFORE INSERT OR UPDATE OF salary, job_id ON hr.employees
   FOR EACH ROW
   WHEN (new.job_id <> 'AD_VP')
   CALL check_sal(:new.job_id, :new.salary, :new.last_name);
```

The procedure check_sal could be implemented in PL/SQL, C, or Java. Also, you can specify :OLD values in the CALL clause instead of :NEW values.

Creating a Database Event Trigger: Example This example creates a trigger to log all errors. The hypothetical PL/SQL block does some special processing for a particular error (invalid logon, error number 1017). This trigger is an AFTER statement trigger, so it is fired after an unsuccessful statement execution (such as unsuccessful logon).

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
   BEGIN
      IF (IS_SERVERERROR (1017)) THEN
         <special processing of logon error>
      ELSE
         <log error number>
      END IF;
   END;
```

Creating an INSTEAD OF Trigger: Example In this example, an oe.order_info view is created to display information about customers and their orders:

```
CREATE VIEW order_info AS
   SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
          o.order_id, o.order_date, o.order_status
   FROM customers c, orders o
   WHERE c.customer_id = o.customer_id;
```

Normally this view would not be updatable, because the primary key of the orders table (order_id) is not unique in the result set of the join view. To make this view updatable, create an INSTEAD OF trigger on the view to process INSERT statements directed to the view (the PL/SQL trigger implementation is shown in italics):

```
CREATE OR REPLACE TRIGGER order_info_insert
   INSTEAD OF INSERT ON order_info
   DECLARE
```

```
    duplicate_info EXCEPTION;
    PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
  BEGIN
    INSERT INTO customers
      (customer_id, cust_last_name, cust_first_name)
    VALUES (
    :new.customer_id,
    :new.cust_last_name,
    :new.cust_first_name);
  INSERT INTO orders
    (order_id, order_date, customer_id)
  VALUES (
    :new.order_id,
    :new.order_date,
    :new.customer_id);
  EXCEPTION
    WHEN duplicate_info THEN
      RAISE_APPLICATION_ERROR (
        num=> -20107,
        msg=> 'Duplicate customer or order ID');
  END order_info_insert;
/
```

You can now insert into both base tables through the view (as long as all NOT NULL columns receive values):

```
INSERT INTO order_info VALUES
   (999, 'Smith', 'John', 2500, '13-MAR-2001', 0);
```

Creating a SCHEMA Trigger: Example The following example creates a BEFORE statement trigger on the sample schema hr. Whenever a user connected as hr attempts to drop any database object, Oracle fires the trigger before dropping the object:

```
CREATE OR REPLACE TRIGGER drop_trigger
   BEFORE DROP ON hr.SCHEMA
   BEGIN
      RAISE_APPLICATION_ERROR (
         num => -20000,
         msg => 'Cannot drop object');
   END;
/
```

16

SQL Statements: CREATE TYPE to DROP ROLLBACK SEGMENT

This chapter contains the following SQL statements:

- CREATE TYPE
- CREATE TYPE BODY
- CREATE USER
- CREATE VIEW
- DELETE
- DISASSOCIATE STATISTICS
- DROP CLUSTER
- DROP CONTEXT
- DROP DATABASE LINK
- DROP DIMENSION
- DROP DIRECTORY
- DROP FUNCTION
- DROP INDEX
- DROP INDEXTYPE
- DROP JAVA
- DROP LIBRARY
- DROP MATERIALIZED VIEW

- DROP MATERIALIZED VIEW LOG
- DROP OPERATOR
- DROP OUTLINE
- DROP PACKAGE
- DROP PROCEDURE
- DROP PROFILE
- DROP ROLE
- DROP ROLLBACK SEGMENT

CREATE TYPE

Purpose

Use the CREATE TYPE statement to create the specification of an **object type**, a **SQLJ object type** (which is a kind of object type), a named varying array (**varray**), a **nested table type**, or an **incomplete object type**. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods in the type.

Notes:

- If you create an object type for which the type specification declares only attributes but no methods, you need not specify a type body.
- If you create a SQLJ object type, you cannot specify a type body. The implementation of the type is specified as a Java class.

Oracle implicitly defines a constructor method for each user-defined type that you create. A **constructor** is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the same as the name of the user-defined type. You can also create a user-defined constructor using the *constructor_spec* syntax.

The parameters of the object type constructor method are the data attributes of the object type. They occur in the same order as the attribute definition order for the object type. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

An **incomplete type** is a type created by a forward type definition. It is called "incomplete" because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

See Also:

- CREATE TYPE BODY on page 16-25 for information on creating the member methods of a type
- *PL/SQL User's Guide and Reference*, *Oracle9i Application Developer's Guide - Object-Relational Features*, and *Oracle9i Database Concepts* for more information about objects, incomplete types, varrays, and nested tables

Prerequisites

To create a type in your own schema, you must have the CREATE TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. You can acquire these privileges explicitly or be granted them through a role.

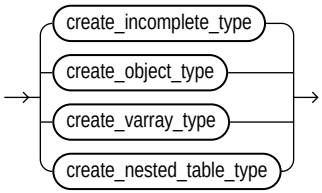
To create a subtype, you must have the UNDER ANY TYPE system privilege or the UNDER object privilege on the supertype.

The owner of the type must either be explicitly granted the EXECUTE object privilege in order to access all other types referenced within the definition of the type, or the type owner must be granted the EXECUTE ANY TYPE system privilege. The owner *cannot* obtain these privileges through roles.

If the type owner intends to grant other users access to the type, the owner must be granted the EXECUTE object privilege to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

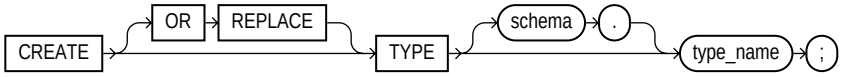
Syntax

create_type::=

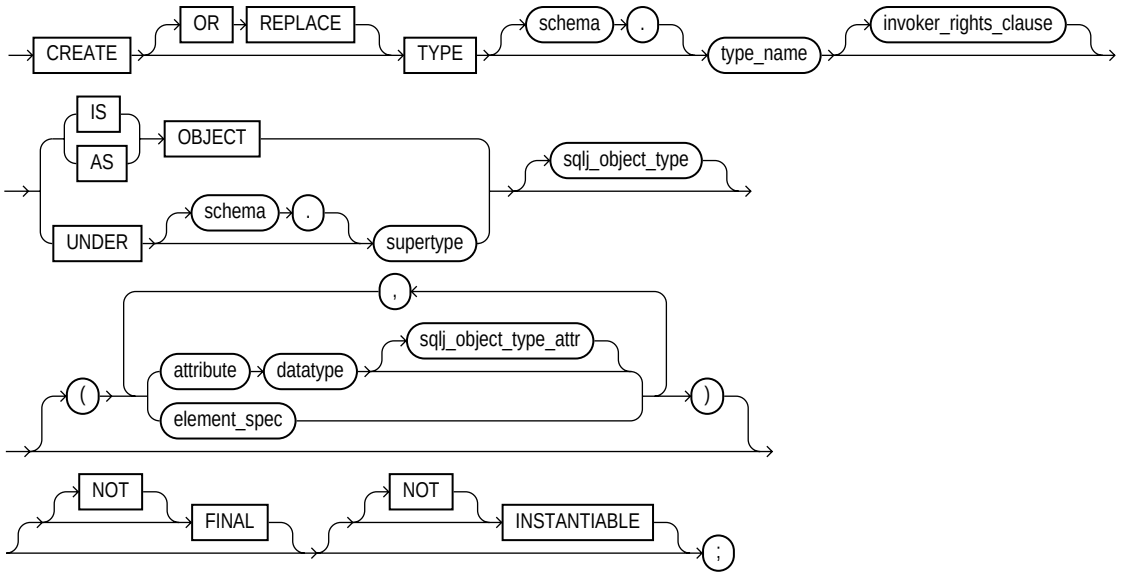


(*create_incomplete_type::=* on page 16-5, *create_object_type::=* on page 16-5, *create_varray_type::=* on page 16-8, *create_nested_table_type::=* on page 16-9)

create_incomplete_type::=

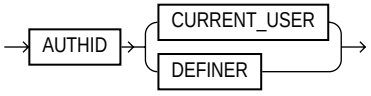


create_object_type::=

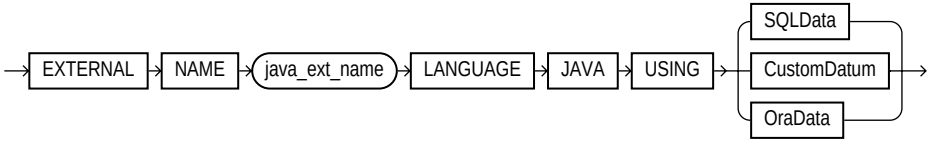


(*invoker_rights_clause::=* on page 16-5, *element_spec::=* on page 16-6)

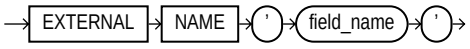
invoker_rights_clause::=



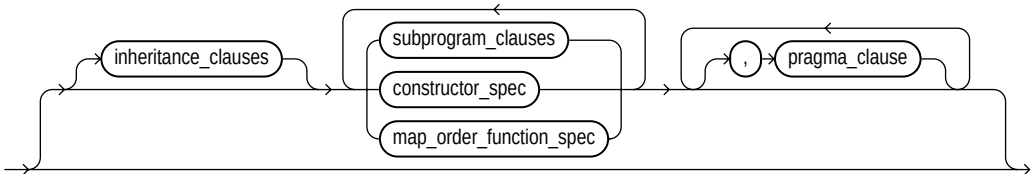
sqlj_object_type::=



sqlj_object_type_attr::=

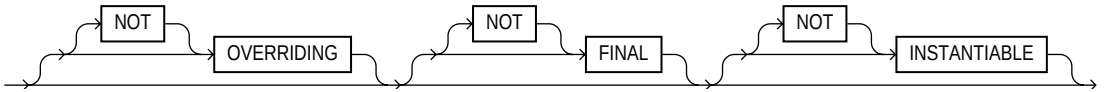


element_spec::=

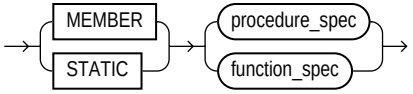


(*inheritance_clauses::=* on page 16-6, *subprogram_clauses::=* on page 16-6, *constructor_spec::=* on page 16-7, *map_order_function_spec::=* on page 16-7, *pragma_clause::=* on page 16-8)

inheritance_clauses::=

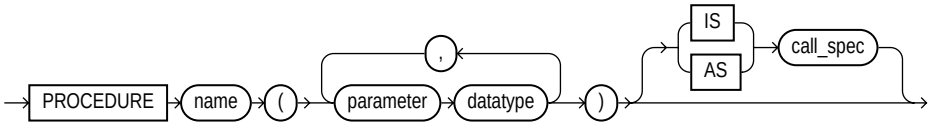


subprogram_clauses::=



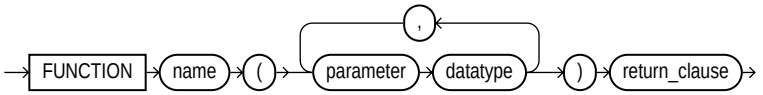
(*procedure_spec::=* on page 16-6, *function_spec::=* on page 16-6)

procedure_spec::=



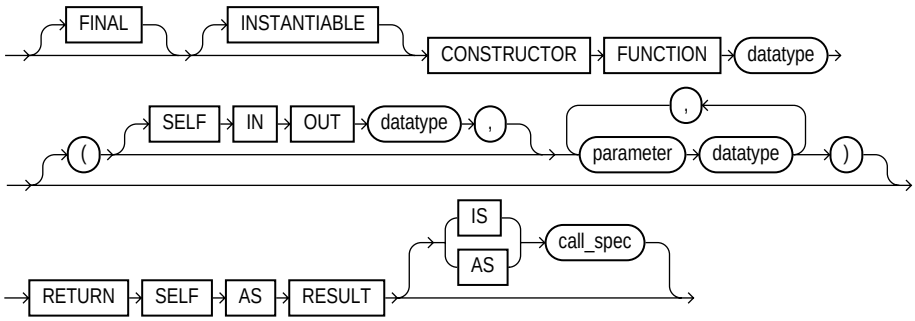
(*call_spec::=* on page 16-8)

function_spec::=



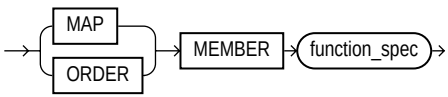
(*return_clause::=* on page 16-7)

constructor_spec::=



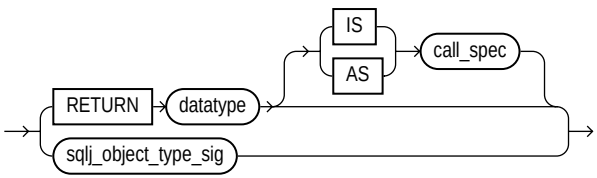
(*call_spec::=* on page 16-8)

map_order_function_spec::=



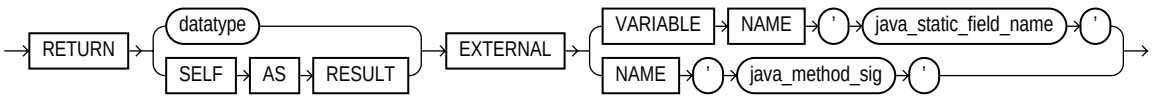
(*function_spec::=* on page 16-6)

return_clause::=

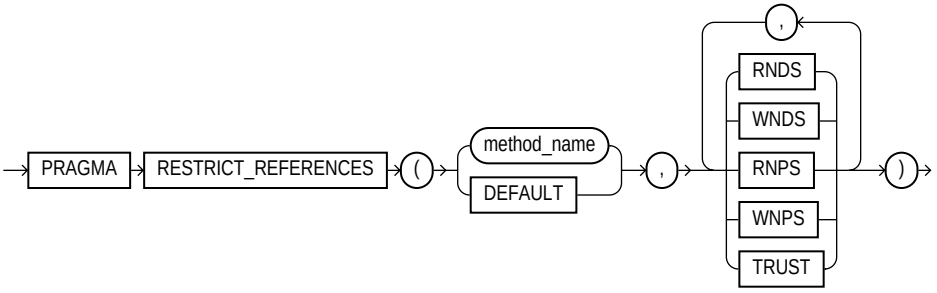


(*call_spec::=* on page 16-8, *sqlj_object_type_sig::=* on page 16-7)

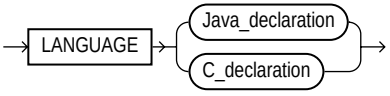
sqlj_object_type_sig::=



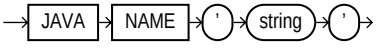
pragma_clause::=



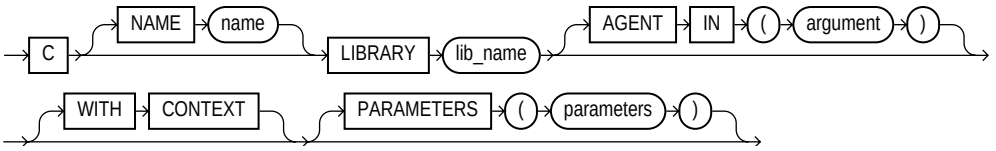
call_spec::=



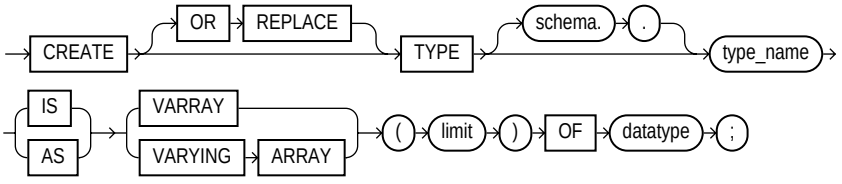
Java_declaration::=



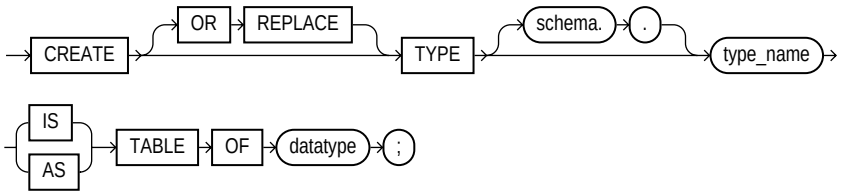
C_declaration::=



create_varray_type::=



create_nested_table_type::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the type if it already exists. Use this clause to change the definition of an existing type without first dropping it.

Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again.

If any function-based indexes depend on the type, Oracle marks the indexes DISABLED.

schema

Specify the schema to contain the type. If you omit *schema*, Oracle creates the type in your current schema.

type_name

Specify the name of an object type, a nested table type, or a varray type.

If creating the type results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

create_object_type

Use the *create_object_type* clause to create a user-defined object type (rather than an incomplete type). The variables that form the data structure are called **attributes**. The member subprograms that define the object's behavior are called **methods**. The keywords AS OBJECT are required when creating an object type.

See Also: "Object Type Examples" on page 16-19

invoker_rights_clause

The *invoker_rights_clause* lets you specify whether the member functions and procedures of the object type execute with the privileges and in the schema of the user who owns the object type or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding type body as well.

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type.

- Specify AUTHID CURRENT_USER if you want the member functions and procedures of the object type to execute with the privileges of CURRENT_USER. This clause creates an **invoker-rights type**.

  This clause also indicates that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the type resides.

- Specify AUTHID DEFINER if you want the member functions and procedures of the object type to execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default and creates a **definer-rights type**.

Restrictions on the *invoker_rights_clause***:**

- You can specify this clause only for an object type, not for a nested table or varray type.
- You can specify this clause for clarity if you are creating a subtype. However, subtypes inherit the rights model of their supertypes, so you cannot specify a different value than was specified for the supertype.
- If the supertype was created with definer's rights, you must create the subtype in the same schema as the supertype.

See Also:

- *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals* for information on how CURRENT_USER is determined
- *PL/SQL User's Guide and Reference*

AS OBJECT Clause

Specify AS OBJECT to create a top-level (root) object type.

UNDER Clause

Specify UNDER *supertype* to create a subtype of an existing type. The existing supertype must be an object type. The subtype you create in this statement inherits the properties of its supertype, and must either override some of those properties or add new properties to distinguish it from the supertype.

> **See Also:** "Subtype Example" on page 16-20 and "Type Hierarchy Example" on page 16-22

sqlj_object_type

Specify the this clause to create a **SQLJ object type**. In a SQLJ object type, you map a Java class to a SQL user-defined type. You can then define tables or columns on SQLJ object type as you would with any other user-defined type.

You can map one Java class to multiple SQLJ object types. If there exists a subtype or supertype of a SQLJ object type, it must be a SQLJ object type. That is, all types in the hierarchy must be SQLJ object types.

java_ext_name Specify the name of the Java class. If the class exists, it must be public. The Java external name, including the schema, will be validated.

Multiple SQLJ object types can be mapped to the same class. However:

- A subtype must be mapped to a class that is an immediate subclass of the class to which its supertype is mapped.
- Two subtypes of a common supertype cannot be mapped to the same class.

SQLData | CustomDatum | OraData Choose the mechanism for creating the Java instance of the type. SQLData, CustomDatum, and OraData are the interfaces that determine which mechanism will be used.

> **See Also:** *Oracle9i JDBC Developer's Guide and Reference* for information on these three interfaces and "SQLJ Object Type Example" on page 16-20

element_spec

The *element_spec* lets you specify each attribute of the object type.

attribute

For *attribute*, specify the name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object. You must specify at least one attribute for each object type.

If you are creating a subtype, the attribute name cannot be the same as any attribute or method name declared in the supertype chain.

datatype

For *datatype*, specify the Oracle built-in datatype or user-defined type of the attribute.

Datatype restrictions:

- You cannot specify attributes of type ROWID, LONG, or LONG ROW.
- You cannot specify a datatype of UROWID for a user-defined object type.
- If you specify an object of type REF, then the target object must have an object identifier.
- If you are creating a collection type for use as a nested table or varray column of a table, then you cannot specify attributes of type AnyType, AnyData, or AnyDataSet.

See Also: "Datatypes" on page 2-2 for a list of valid datatypes

sqlj_object_type_attr

This clause is valid only if you have specified the *sqlj_object_type* clause (that is, you are mapping a Java class to a SQLJ object type). Specify the external name of the Java field that corresponds to the attribute of the SQLJ object type. The Java *field_name* must already exist in the class. You cannot map a Java *field_name* to more than one SQLJ object type attribute in the same type hierarchy.

This clause is optional when you create a SQLJ object type.

subprogram_clauses

The *subprogram_clauses* let you associate a procedure subprogram with the object type.

MEMBER Clause

Specify a function or procedure subprogram associated with the object type that is referenced as an attribute. Typically, you invoke MEMBER methods in a "selfish"

style, such as `object_expression.method()`. This class of method has an implicit first argument referenced as `SELF` in the method's body, which represents the object on which the method has been invoked.

Restriction on member methods: You cannot specify a `MEMBER` method if you are mapping a Java class to a `SQLJ` object type.

See Also: "Creating a Member Method: Example" on page 16-23

STATIC Clause

Specify a function or procedure subprogram associated with the object type. Unlike `MEMBER` methods, `STATIC` methods do not have any implicit parameters (that is, you cannot reference `SELF` in their body). They are typically invoked as `type_name.method()`.

Restrictions on static methods:

- You cannot map a `MEMBER` method in a Java class to a `STATIC` method in a `SQLJ` object type.
- For both `MEMBER` and `STATIC` methods, you must specify a corresponding method body in the object type body for each procedure or function specification.

See Also: "Creating a Static Method: Example" on page 16-24

[NOT] FINAL, [NOT] INSTANTIABLE

At the top level of the syntax, these clauses specify the inheritance attributes of the type.

Use the [`NOT`] `FINAL` clause to indicate whether any further subtypes can be created for this type:

- Specify `FINAL` if no further subtypes can be created for this type. This is the default.
- Specify `NOT FINAL` if further subtypes can be created under this type.

Use the [`NOT`] `INSTANTIABLE` clause to indicate whether any object instances of this type can be constructed:

- Specify `INSTANTIABLE` if object instances of this type can be constructed. This is the default.
- Specify `NOT INSTANTIABLE` if no constructor (default or user-defined) exists for this object type. You must specify these keywords for any type with

noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement). You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement).

inheritance_clauses

As part of the *element_spec*, the *inheritance_clauses* let you specify the relationship between super- and subtypes.

OVERRIDING This clause is valid only for MEMBER methods. Specify OVERRIDING to indicate that this method overrides a MEMBER method defined in the supertype. This keyword is required if the method redefines a supertype method. NOT OVERRIDING is the default.

Restriction: The OVERRIDING clause is not valid for a STATIC method or for a SQLJ object type.

FINAL Specify FINAL to indicate that this method cannot be overridden by any subtype of this type. The default is NOT FINAL.

NOT INSTANTIABLE Specify NOT INSTANTIABLE if the type does not provide an implementation for this method. By default all methods are INSTANTIABLE.

Restriction on NOT INSTANTIABLE: If you specify NOT INSTANTIABLE, you cannot specify FINAL or STATIC.

See Also: *constructor_spec* on page 16-16

procedure_spec or *function_spec*

Use these clauses to specify the parameters and datatypes of the procedure or function. If this subprogram does not include the declaration of the procedure or function, you must issue a corresponding CREATE TYPE BODY statement.

Restriction on procedure and function specification: If you are creating a subtype, the name of the procedure or function cannot be the same as the name of any attribute, whether inherited or not, declared in the supertype chain.

return_clause The first form of the *return_clause* is valid only for a function. The syntax shown is an abbreviated form.

See Also:

- *PL/SQL User's Guide and Reference* for information about method invocation and methods
- CREATE PROCEDURE on page 14-62 and CREATE FUNCTION on page 13-49 for the full syntax with all possible clauses
- CREATE TYPE BODY on page 16-25
- "Restrictions on user-defined functions:" on page 13-53 for a list of restrictions on user-defined functions

sqlj_object_type_sig Use this form of the *return_clause* if you intend to create SQLJ object type functions or procedures.

- If you are mapping a Java class to a SQLJ object type and you specify EXTERNAL NAME, the value of the Java method returned must be compatible with the SQL returned value, and the Java method must be public. Also, the method signature (method name plus parameter types) must be unique within the type hierarchy.
- If you specify EXTERNAL VARIABLE NAME, the type of the Java static field must be compatible with the return type.

call_spec

Specify the call specification ("call spec") that maps a Java or C method name, parameter types, and return type to their SQL counterparts. If all the member methods in the type have been defined in this clause, you need not issue a corresponding CREATE TYPE BODY statement.

The *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle9i Java Stored Procedures Developer's Guide*.
- *Oracle9i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

pragma_clause

The *pragma_clause* lets you specify a compiler directive. The PRAGMA RESTRICT_REFERENCES compiler directive denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

Note: Oracle Corporation recommends that you avoid using this clause unless you must do so for backward compatibility of your applications. This clause has been deprecated, because beginning with Oracle9*i*, Oracle runs purity checks at run time.

---

method Specify the name of the MEMBER function or procedure to which the pragma is being applied.

DEFAULT Specify DEFAULT if you want Oracle to apply the pragma to all methods in the type for which a pragma has not been explicitly specified.

WNDS Specify WNDS to enforce the constraint writes no database state (does not modify database tables).

WNPS Specify WNPS to enforce the constraint writes no package state (does not modify packaged variables).

RNDS Specify RNDS to enforce the constraint reads no database state (does not query database tables).

RNPS Specify WNPS to enforce the constraint reads no package state (does not reference package variables).

TRUST Specify TRUST to indicate that the restrictions listed in the pragma are not actually to be enforced, but are simply trusted to be true.

See Also: *Oracle9i Application Developer's Guide - Fundamentals*

constructor_spec

Use this clause to create a user-defined constructor, which is a function that returns an initialized instance of a user-defined object type. You can declare multiple constructors for a single object type, as long as the parameters of each constructor differ in number, order, or datatype.

- User-defined constructor functions are always FINAL and INSTANTIABLE, so these keywords are optional.
- The parameter-passing mode of user-defined constructors is always SELF IN OUT. Therefore you need not specify this clause unless you wish to do so explicitly for clarity.
- RETURN SELF AS RESULT specifies that the runtime type of the value returned by the constructor is the same as the runtime type of the SELF argument.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for more information on and examples of user-defined constructors and "Constructor Example" on page 16-23

map_order_function_spec

You can define either a MAP method or an ORDER method in a type specification, but not both. Also, you cannot define either MAP or ORDER methods for subtypes. However, a subtype can override a MAP method if the supertype defines a nonfinal MAP method. (A subtype cannot override an ORDER method at all.) If you declare either method, you can compare object instances in SQL.

You can specify either MAP or ORDER when mapping a Java class to a SQL type. However, the MAP or ORDER methods must map to MEMBER functions in the Java class.

If neither a MAP nor an ORDER method is specified, only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types.

Use MAP if you are performing extensive sorting or hash join operations on object instances. MAP is applied once to map the objects to scalar values and then the scalars are used during sorting and merging. A MAP method is more efficient than an ORDER method, which must invoke the method for each object comparison. You must use a MAP method for hash joins. You cannot use an ORDER method because the hash mechanism hashes on the object value.

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information about object value comparisons

MAP MEMBER This clause lets you specify a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the

object. A MAP method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the MAP method is null, the MAP method returns null and the method is not invoked.

An object specification can contain only one MAP method, which must be a function. The result type must be a predefined SQL scalar type, and the MAP method can have no arguments other than the implicit SELF argument.

Note: If *type_name* will be referenced in queries involving sorts (through an ORDER BY, GROUP BY, DISTINCT, or UNION clause) or joins, and you want those queries to be parallelized, you must specify a MAP member function.

A subtype cannot define a new MAP method. However it can override an inherited MAP method.

ORDER MEMBER This clause lets you specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the ORDER method is null, the ORDER method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the ORDER method *map_order_function_spec* is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

A subtype can neither define nor override an ORDER method.

create_varray_type

The *create_varray_type* lets you create the type as an ordered set of elements, each of which has the same datatype. You must specify a name and a maximum limit of zero or more. The array limit must be an integer literal. Oracle does not support anonymous varrays.

The type name for the objects contained in the varray must be one of the following:

- A built-in datatype,
- A REF, or
- An object type.

Restrictions on varray types:

- You cannot create varray types of LOB datatypes.
- You can create a VARRAY type of XMLType for use in PL/SQL or in view queries. However, you cannot create a column of this varray type, because Oracle stores XMLType data as CLOB (see preceding restriction).

See Also: "Varray Type Example" on page 16-22

create_nested_table_type

The *create_nested_table_type* lets you create a named nested table of type *datatype*.

- When *datatype* is an object type, the nested table type describes a table whose columns match the name and attributes of the object type.
- When *datatype* is a scalar type, then the nested table type describes a table with a single, scalar type column called "column_value".

Restriction on nested table types: You cannot specify NCLOB for *datatype*. However, you can specify CLOB or BLOB.

See Also: "Named Table Type Example" on page 16-22 and "Nested Table Type Containing a Varray" on page 16-22

Examples

Object Type Examples The following example shows how the sample type customer_typ was created for the sample Order Entry (oe) schema:

```
CREATE TYPE customer_typ_demo AS OBJECT
    ( customer_id        NUMBER(6)
    , cust_first_name    VARCHAR2(20)
    , cust_last_name     VARCHAR2(20)
    , cust_address       CUST_ADDRESS_TYP
    , phone_numbers      PHONE_LIST_TYP
    , nls_language       VARCHAR2(3)
    , nls_territory      VARCHAR2(30)
    , credit_limit       NUMBER(9,2)
```

```
    , cust_email         VARCHAR2(30)
    , cust_orders        ORDER_LIST_TYP
    ) ;
```

In the following example, the `data_typ` object type is created with one member function `prod`, which is implemented in the `CREATE TYPE BODY` statement:

```
CREATE TYPE data_typ AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );

CREATE TYPE BODY data_typ IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
         BEGIN
             RETURN (year + invent);
         END;
      END;
```

Subtype Example The following statement shows how the subtype `corporate_customer_typ` in the sample `oe` schema was created. It is based on the `customer_typ` supertype created in the preceding example and adds the `account_mgr_id` attribute:

```
CREATE TYPE corporate_customer_typ_demo UNDER customer_typ
    ( account_mgr_id     NUMBER(6)
    );
```

SQLJ Object Type Example The following examples create a SQLJ object type and subtype. The `address_t` type maps to the Java class `Examples.Address`. The subtype `long_address_t` maps to the Java class `Examples.LongAddress`. The examples specify SQLData as the mechanism used to create the Java instance of these types. Each of the functions in these type specifications have corresponding implementations in the Java class.

> **See Also:** *Oracle9i Application Developer's Guide - Object-Relational Features* for the Java implementation of the functions in these type specifications

```
CREATE TYPE address_t AS OBJECT
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
```

```
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME 'create (java.lang.String, java.lang.String,
java.lang.String, int) return Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME
        'create (java.lang.String, java.lang.String,
java.lang.String, int) return Examples.Address',
    MEMBER FUNCTION to_string RETURN VARCHAR
      EXTERNAL NAME 'tojava.lang.String() return java.lang.String',
    MEMBER FUNCTION strip RETURN SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks () return Examples.Address'
  ) NOT FINAL;

CREATE OR REPLACE TYPE long_address_t
UNDER address_t
EXTERNAL NAME 'Examples.LongAddress' LANGUAGE JAVA
USING SQLData(
    street2_attr VARCHAR(250) EXTERNAL NAME 'street2',
    country_attr VARCHAR (200) EXTERNAL NAME 'country',
    address_code_attr VARCHAR (50) EXTERNAL NAME 'addrCode',
    STATIC FUNCTION create_address RETURN long_address_t
      EXTERNAL NAME 'create() return Examples.LongAddress',
    STATIC FUNCTION  construct (street VARCHAR, city VARCHAR,
        state VARCHAR, country VARCHAR, addrs_cd VARCHAR)
      RETURN long_address_t
      EXTERNAL NAME
        'create(java.lang.String, java.lang.String,
        java.lang.String, java.lang.String, java.lang.String)
          return Examples.LongAddress',
    STATIC FUNCTION construct RETURN long_address_t
      EXTERNAL NAME 'Examples.LongAddress()
        return Examples.LongAddress',
    STATIC FUNCTION create_longaddress (
      street VARCHAR, city VARCHAR, state VARCHAR, country VARCHAR,
      addrs_cd VARCHAR) return long_address_t
```

```
      EXTERNAL NAME
        'Examples.LongAddress (java.lang.String, java.lang.String,
         java.lang.String, java.lang.String, java.lang.String)
           return Examples.LongAddress',
    MEMBER FUNCTION get_country RETURN VARCHAR
      EXTERNAL NAME 'country_with_code () return java.lang.String'
  );
```

Type Hierarchy Example The following statements creates a type hierarchy. Type `employee_t` inherits the `name` and `ssn` attributes from type `person_t`, and in addition has `dept_id` and `salary` attributes. Type `part_time_emp_t` inherits all of the attributes from `employee_t` and, through `employee_t`, those of `person_t`, and in addition has a `num_hrs` attribute. Type `part_time_emp_t` is final by default, so no further subtypes can be created under it.:

```
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;

CREATE TYPE employee_t UNDER person_t
   (dept_id NUMBER, salary NUMBER) NOT FINAL;

CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
```

You can use type hierarchies to create substitutable tables and tables with substitutable columns. For examples, see "Substitutable Table and Column Examples" on page 15-67.

Varray Type Example The following statement shows how the `phone_list_typ` varray type with 5 elements in the sample `oe` schema was created:

```
CREATE TYPE phone_list_typ_demo AS VARRAY(5) OF VARCHAR2(25);
```

Named Table Type Example The following example from the sample schema `pm` creates the named table type `textdoc_tab` of object type `textdoc_typ`:

```
CREATE TYPE textdoc_typ AS OBJECT
    ( document_typ      VARCHAR2(32)
    , formatted_doc     BLOB
    ) ;

CREATE TYPE textdoc_tab AS TABLE OF textdoc_typ;
```

Nested Table Type Containing a Varray The following example of multilevel collections is a variation of the sample table `oe.customers`. In this example, the

cust_address object column becomes a nested table column with the phone_list_typ varray column embedded in it:

```
CREATE TYPE phone_list_typ AS VARRAY(5) OF VARCHAR2(25);

CREATE TYPE cust_address_typ2 AS OBJECT
       ( street_address     VARCHAR2(40)
       , postal_code        VARCHAR2(10)
       , city               VARCHAR2(30)
       , state_province     VARCHAR2(10)
       , country_id         CHAR(2)
       , phone              phone_list_typ
       );

CREATE TYPE cust_nt_address_typ
   AS TABLE OF cust_address_typ;
```

Constructor Example This example invokes the system-defined constructor to construct the demo_typ object and insert it into the demo_tab table:

```
CREATE TYPE demo_typ1 AS OBJECT (a1 NUMBER, a2 NUMBER);

CREATE TABLE demo_tab1 (b1 NUMBER, b2 demo_typ1);

INSERT INTO demo_tab1 VALUES (1, demo_typ1(2,3));
```

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* and *PL/SQL User's Guide and Reference* for more information about constructors

Creating a Member Method: Example The following example invokes method constructor col.getbar(). (The example assumes the getbar method already exists.)

```
CREATE TYPE demo_typ2 AS OBJECT (a1 NUMBER,
   MEMBER FUNCTION getbar RETURN NUMBER);

CREATE TABLE demo_tab2(col demo_typ2);

SELECT col.getbar() FROM demo_tab2;
```

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

Creating a Static Method: Example The following example changes the definition of the `employee_t` type to associate it with the `construct_emp` function. The example first creates an object type `department_t` and then an object type `employee_t` containing an attribute of type `department_t`:

```
CREATE OR REPLACE TYPE department_t AS OBJECT (
   deptno number(10),
   dname CHAR(30));

CREATE OR REPLACE TYPE employee_t AS OBJECT(
   empid RAW(16),
   ename CHAR(31),
   dept REF department_t,
      STATIC function construct_emp
      (name VARCHAR2, dept REF department_t)
      RETURN employee_t
);
```

This statement requires the following type body statement (PL/SQL is shown in italics):

```
CREATE OR REPLACE TYPE BODY employee_t IS
   STATIC FUNCTION construct_emp
   (name varchar2, dept REF department_t)
   RETURN employee_t IS
      BEGIN
         return employee_t(SYS_GUID(),name,dept);
      END;
END;
```

Next create an object table and insert into the table:

```
CREATE TABLE emptab OF employee_t;
INSERT INTO emptab
   VALUES (employee_t.construct_emp('John Smith', NULL));
```

CREATE TYPE BODY

Purpose

Use the CREATE TYPE BODY to define or implement the member methods defined in the object type specification. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods in the type.

For each method specified in an object type specification for which you did not specify the *call_spec,* you must specify a corresponding method body in the object type body.

Note: If you create a SQLJ object type, you cannot specify a type body. The implementation of the type is specified as a Java class.

See Also: CREATE TYPE on page 16-3 and ALTER TYPE on page 12-6 for information on creating and modifying a type specification

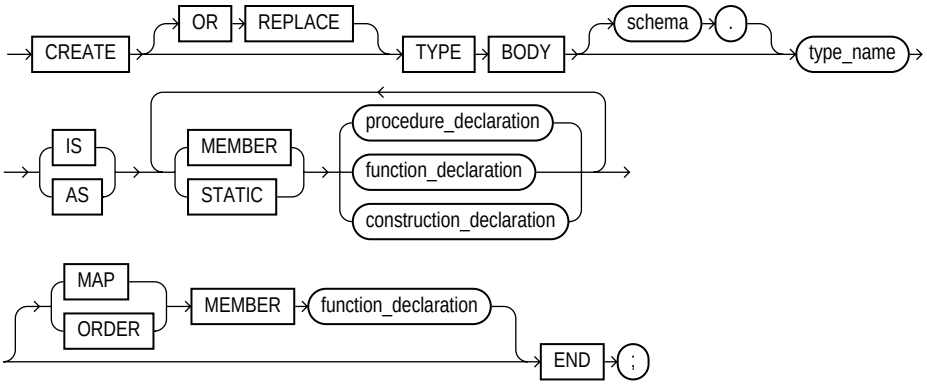
Prerequisites

Every member declaration in the CREATE TYPE specification for object types must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create an object type in another user's schema, you must have the CREATE ANY TYPE system privileges. To replace an object type in another user's schema, you must have the DROP ANY TYPE system privileges.

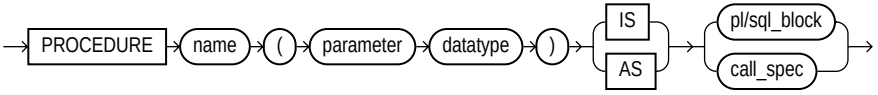
Syntax

create_type_body::=



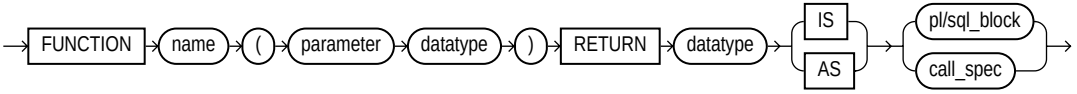
(*procedure_declaration::=* on page 16-26, *function_declaration::=* on page 16-26, *constructor_declaration::=* on page 16-27)

procedure_declaration::=



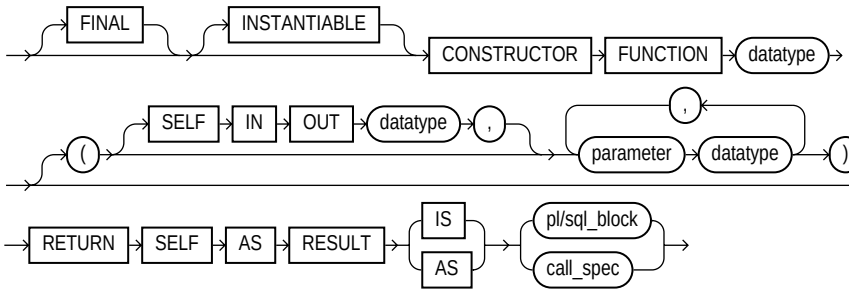
(*call_spec::=* on page 16-8)

function_declaration::=

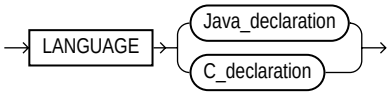


(*call_spec::=* on page 16-8)

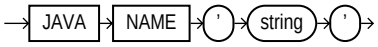
constructor_declaration::=



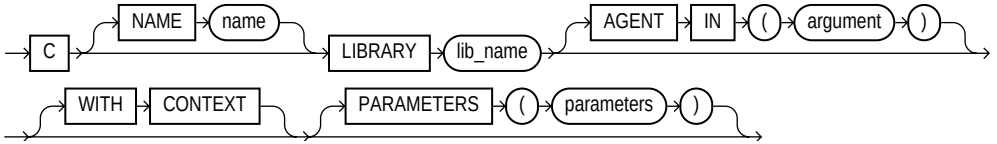
call_spec::=



Java_declaration::=



C_declaration::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the type body if it already exists. Use this clause to change the definition of an existing type body without first dropping it.

Users previously granted privileges on the re-created object type body can use and reference the object type body without being granted privileges again.

You can use this clause to add new member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE statement.

schema

Specify the schema to contain the type body. If you omit *schema*, Oracle creates the type body in your current schema.

type_name

Specify the name of an object type.

IS | AS

MEMBER | STATIC

Specify the type of method function or procedure subprogram associated with the object type specification.

You must define a corresponding method name, optional parameter list, and (for functions) a return type in the object type specification for each procedure or function declaration.

procedure_declaration, function_declaration, Declare a procedure or function subprogram.

constructor_declaration Declare a user-defined constructor subprogram. The RETURN clause of a constructor function must be RETURN SELF AS RESULT. This setting indicates that the most specific type of the value returned by the constructor function is the same as the most specific type of the SELF argument that was passed in to the constructor function.

See Also:

- CREATE TYPE on page 16-3 for a list of restrictions on user-defined functions
- *PL/SQL User's Guide and Reference* for information about overloading subprogram names within a package
- CREATE PROCEDURE on page 14-62, CREATE FUNCTION on page 13-49, and *Oracle9i Application Developer's Guide - Fundamentals* for information on the components of type body
- *Oracle9i Application Developer's Guide - Object-Relational Features* for information on and examples of user-defined constructors

pl/sql_block Declare the procedure or function.

See Also: *PL/SQL User's Guide and Reference*

call_spec Specify the call specification ("call spec") that maps a Java or C method name, parameter types, and return type to their SQL counterparts.

The *Java_declaration*, '*string*' identifies the Java implementation of the method.

See Also:

- *Oracle9i Java Stored Procedures Developer's Guide*
- *Oracle9i Application Developer's Guide - Fundamentals* for an explanation of the parameters and semantics of the *C_declaration*

MAP | ORDER Method

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

MAP MEMBER Clause

Specify MAP MEMBER to declare or implement a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A MAP method is called implicitly and specifies an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the MAP method is null, the MAP method returns null and the method is not invoked.

An object type body can contain only one MAP method, which must be a function. The MAP function can have no arguments other than the implicit SELF argument.

ORDER MEMBER Clause

Specify ORDER MEMBER to specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the ORDER method is null, the ORDER method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, Oracle invokes the ORDER MEMBER *function_declaration*.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

function_declaration Declare a function subprogram.

> **See Also:** CREATE PROCEDURE on page 14-62 and CREATE FUNCTION on page 13-49 for the full syntax with all possible clauses

AS EXTERNAL AS EXTERNAL is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the *call_spec* syntax with the *C_declaration*.

Examples

Several examples of creating type bodies appear in the "Examples" section of CREATE TYPE on page 16-19.

Updating a Type Body: Example The following example shows how the type body of the data_typ object type (see "Object Type Examples" on page 16-19) must be modified when an attribute is added to the type:

```
ALTER TYPE data_typ
   ADD MEMBER FUNCTION qtr(der_qtr DATE)
   RETURN CHAR CASCADE;

CREATE OR REPLACE TYPE BODY data_typ IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
  BEGIN
  RETURN (year + invent);
  END;
     MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
     BEGIN
       IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
         RETURN 'FIRST';
       ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
         RETURN 'SECOND';
       ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
```

```
          RETURN 'THIRD';
        ELSE
          RETURN 'FOURTH';
        END IF;
      END;
    END;
/
    END;
```

CREATE USER

Purpose

Use the CREATE USER statement to create and configure a database **user**, or an account through which you can log in to the database and establish the means by which Oracle permits access by the user.

Note: You can enable a user to connect to Oracle through a proxy (that is, an application or application server). For syntax and discussion, refer to ALTER USER on page 12-21.

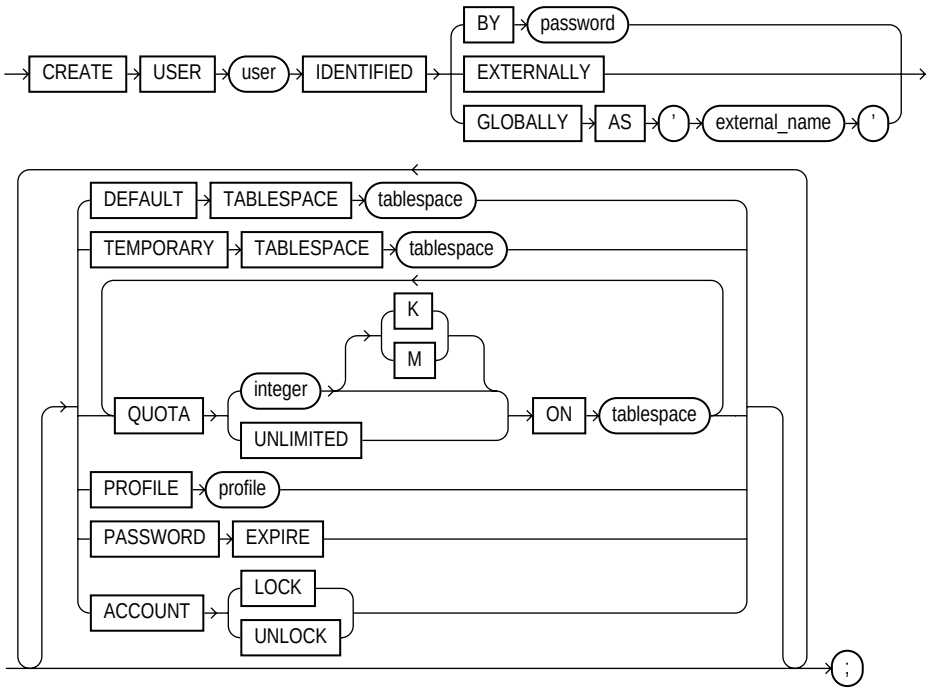
Prerequisites

You must have CREATE USER system privilege. When you create a user with the CREATE USER statement, the user's privilege domain is empty. To log on to Oracle, a user must have CREATE SESSION system privilege. Therefore, after creating a user, you should grant the user at least the CREATE SESSION privilege.

See Also: GRANT on page 17-29

Syntax

create_user::=



Keywords and Parameters

user

Specify the name of the user to be created. This name can contain only characters from your database character set and must follow the rules described in the section "Schema Object Naming Rules" on page 2-111. Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

Note: Oracle Corporation recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform. Please refer to *Oracle9i Database Administrator's Guide* for more information about this recommendation.

See Also: "Creating a Database User: Example" on page 16-37

IDENTIFIED Clause

The `IDENTIFIED` clause lets you indicate how Oracle authenticates the user.

BY password

The `BY` *`password`* clause lets you creates a **local user** and indicates that the user must specify *`password`* to log on. Passwords can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

Passwords must follow the rules described in the section "Schema Object Naming Rules" on page 2-111, unless you are using Oracle's password complexity verification routine. That routine requires a more complex combination of characters than the normal naming rules permit. You implement this routine with the `UTLPWDMG.SQL` script, which is further described in *Oracle9i Database Administrator's Guide.*

Note: Oracle Corporation recommends that user names and passwords be encoded in ASCII or EBCDIC characters only, depending on your platform. Please refer to *Oracle9i Database Administrator's Guide* for more information about this recommendation.

See Also: *Oracle9i Database Administrator's Guide* to for a detailed description and explanation of how to use password management and protection

EXTERNALLY Clause

Specify `EXTERNALLY` to create an **external user.** Such a user must be authenticated by an external service (such as an operating system or a third-party service). In this

case, Oracle to relies on the login authentication of the operating system to ensure that a specific operating system user has access to a specific database user.

Caution: strongly recommends that you do not use `IDENTIFIED EXTERNALLY` with operating systems that have inherently weak login security. For more information, see *Oracle9i Database Administrator's Guide*.

See Also: "Creating External Database Users: Examples" on page 16-38

GLOBALLY Clause

The `GLOBALLY` clause lets you create a **global user**. Such a user must be authenticated by the enterprise directory service. The '*`external_name`*' string can take one of two forms:

- The X.509 name at the enterprise directory service that identifies this user. It should be of the form '`CN=`*`username,other_attributes'`*, where *`other_attributes`* is the rest of the user's distinguished name (DN) in the directory.
- A null string (' ') indicating that the enterprise directory service will map authenticated global users to the appropriate database schema with the appropriate roles.

Note: You can control the ability of an application server to connect as the specified user and to activate that user's roles using the `ALTER USER` statement.

See Also:

- *Oracle Advanced Security Administrator's Guide* for more information on global users
- ALTER USER on page 12-21
- *Oracle9i Application Developer's Guide - Fundamentals* and your operating system specific documentation for more information
- "Creating a Global Database User: Example" on page 16-38

DEFAULT TABLESPACE Clause

Specify the default tablespace for objects that the user creates. If you omit this clause, objects default to the SYSTEM tablespace.

Restriction on default temporary tablespaces: You cannot specify a locally managed tablespace (including an undo tablespace) or a dictionary-managed temporary tablespace as a user's default tablespace.

> **See Also:** CREATE TABLESPACE on page 15-80 for more information on tablespaces in general and undo tablespaces in particular

TEMPORARY TABLESPACE Clause

Specify the tablespace for the user's temporary segments. If you omit this clause, temporary segments default to the SYSTEM tablespace.

Restrictions on a user's temporary tablespaces:

- The tablespace must be a temporary tablespace and must have a standard block size.
- The tablespace cannot be an undo tablespace or a tablespace with automatic segment-space management.

> **See Also:** CREATE TABLESPACE on page 15-80 for more information on undo tablespaces and segment management

QUOTA Clause

Use the QUOTA clause to allow the user to allocate up to *integer* bytes of space in the tablespace. Use K or M to specify the quota in kilobytes or megabytes. This quota is the maximum space in the tablespace the user can allocate.

A CREATE USER statement can have multiple QUOTA clauses for multiple tablespaces.

UNLIMITED lets the user allocate space in the tablespace without bound.

PROFILE Clause

Specify the profile you want to reassign to the user. The profile limits the amount of database resources the user can use. If you omit this clause, Oracle assigns the DEFAULT profile to the user.

See Also: GRANT on page 17-29 and CREATE PROFILE on page 14-69

PASSWORD EXPIRE Clause

Specify PASSWORD EXPIRE if you want the user's password to expire. This setting forces the user (or the DBA) to change the password before the user can log in to the database.

ACCOUNT Clause

Specify ACCOUNT LOCK to lock the user's account and disable access. Specify ACCOUNT UNLOCK to unlock the user's account and enable access to the account.

Examples

Note: All of the following examples use the demo tablespace because it which exists in the seed database and is accessible to the sample schemas.

Creating a Database User: Example If you create a new user with PASSWORD EXPIRE, the user's password must be changed before attempting to log in to the database. You can create the user sidney by issuing the following statement:

```
CREATE USER sidney
    IDENTIFIED BY out_standing1
    DEFAULT TABLESPACE demo
    QUOTA 10M ON demo
    TEMPORARY TABLESPACE temp
    QUOTA 5M ON system
    PROFILE app_user
    PASSWORD EXPIRE;
```

The user sidney has the following characteristics:

- The password welcome
- Default tablespace demo, with a quota of 10 megabytes
- Temporary tablespace temp
- Access to the tablespace SYSTEM, with a quota of 5 megabytes

- Limits on database resources defined by the profile app_user (which was created in "Creating a Profile: Example" on page 14-74)
- An expired password, which must be changed before sidney can log in to the database

Creating External Database Users: Examples The following example creates an external user, who must be identified by an external source before accessing the database:

```
CREATE USER app_user1
   IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE demo
   QUOTA 5M ON demo
   PROFILE app_user;
```

The user app_user1 has the following additional characteristics:

- Default tablespace demo
- Default temporary tablespace demo
- 5M of space on the tablespace demo and unlimited quota on the temporary tablespace of the database
- Limits on database resources defined by the app_user profile

To create another user accessible only by the operating system account app_user2, prefix app_user2 by the value of the initialization parameter OS_AUTHENT_PREFIX. For example, if this value is "ops$", you can create the user ops$app_user2 with the following statement:

```
CREATE USER ops$external_user
   IDENTIFIED EXTERNALLY
   DEFAULT TABLESPACE demo
   QUOTA 5M ON demo
   PROFILE app_user;
```

Creating a Global Database User: Example The following example creates a global user. When you create a global user, you can specify the X.509 name that identifies this user at the enterprise directory server:

```
CREATE USER global_user
   IDENTIFIED GLOBALLY AS 'CN=analyst, OU=division1, O=oracle, C=US'
   DEFAULT TABLESPACE demo
   QUOTA 5M ON demo;
```

CREATE VIEW

Purpose

Use the CREATE VIEW statement to define a **view**, which is a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called **base tables**.

You can also create an **object view** or a relational view that supports LOB and object datatypes (object types, REFs, nested table, or varray types) on top of the existing view mechanism. An object view is a view of a user-defined type, where each row contains objects, each object with a unique object identifier.

You can also create XMLType views, which are similar to an object views but display data from XMLSchema-based tables of XMLType.

See Also:

- *Oracle9i Database Concepts*, *Oracle9i Application Developer's Guide - Fundamentals*, and *Oracle9i Database Administrator's Guide* for information on various types of views and their uses
- *Oracle9i XML Database Developer's Guide - Oracle XML DB* for information on XMLType views
- ALTER VIEW on page 12-30 for information on modifying a view
- DROP VIEW on page 17-22 for information on removing a view from the database

Prerequisites

To create a view in your own schema, you must have CREATE VIEW system privilege. To create a view in another user's schema, you must have CREATE ANY VIEW system privilege.

To create a subview, you must have UNDER ANY VIEW system privilege or the UNDER object privilege on the superview.

The owner of the schema containing the view must have the privileges necessary to either select, insert, update, or delete rows from all the tables or views on which the view is based. The owner must be granted these privileges directly, rather than through a role.

To use the basic constructor method of an object type when creating an object view, one of the following must be true:

- The object type must belong to the same schema as the view to be created.
- You must have EXECUTE ANY TYPE system privileges.
- You must have the EXECUTE object privilege on that object type.

> **See Also:** SELECT on page 18-4, INSERT on page 17-54, UPDATE on page 18-59, and DELETE on page 16-55 for information on the privileges required by the owner of a view on the base tables or views of the view being created

Partition Views

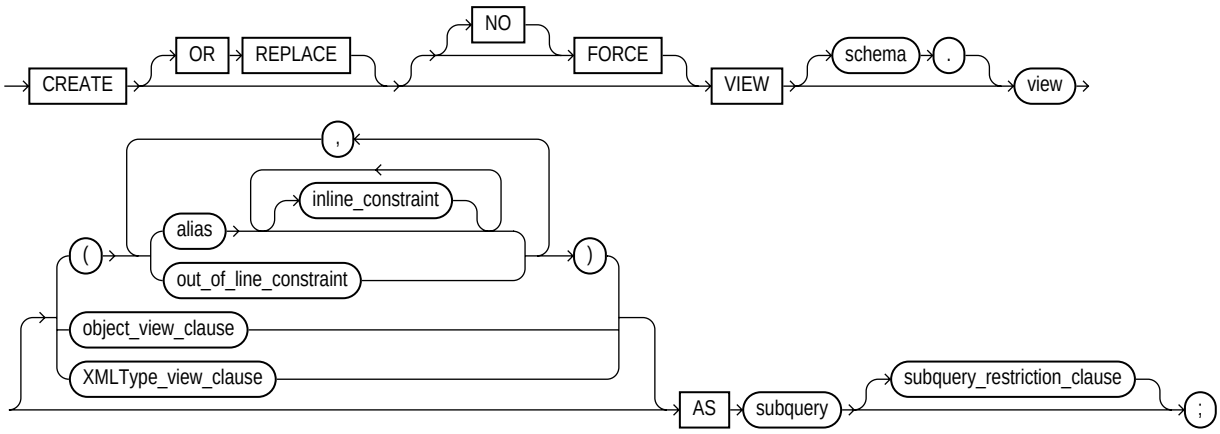
Partition views were introduced in Oracle Release 7.3 to provide partitioning capabilities for applications requiring them. Partition views are supported in Oracle9*i* so that you can upgrade applications from Release 7.3 without any modification. In most cases, subsequent to upgrading to Oracle9*i* you will want to migrate partition views into partitions.

In Oracle9*i*, you can use the CREATE TABLE statement to create partitioned tables easily. Partitioned tables offer the same advantages as partition views, while also addressing their shortcomings. Oracle recommends that you use partitioned tables rather than partition views in most operational environments.

> **See Also:**
>
> - *Oracle9i Database Concepts* for more information on the shortcomings of partition views
> - *Oracle9i Database Administrator's Guide* for information on migrating partition views into partitions
> - CREATE TABLE on page 15-7 for more information about partitioned tables

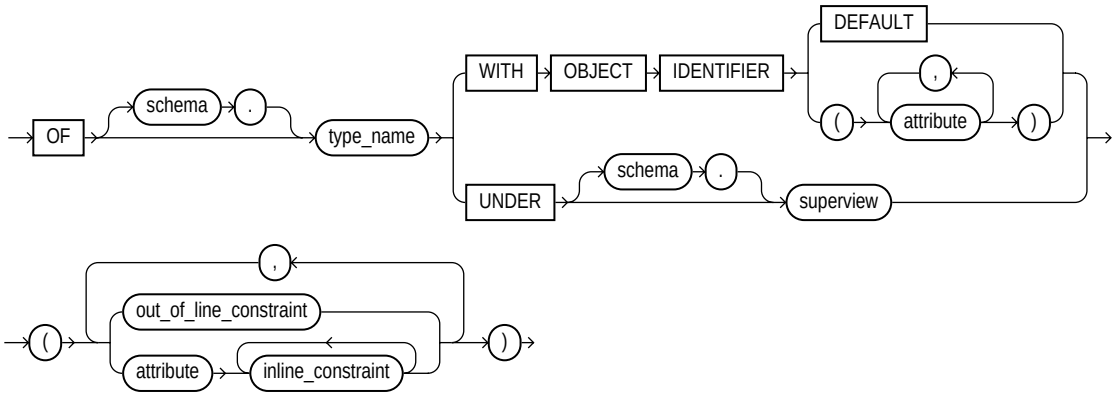
Syntax

create_view::=



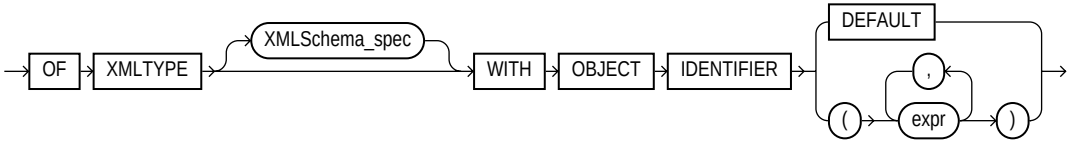
(*inline_constraint::=* on page 7-7 and *out_of_line_constraint::=* on page 7-7—part of *constraints* syntax, *object_view_clause::=* on page 16-41, *XMLType_view_clause::=* on page 16-42, *subquery::=* on page 18-5—part of SELECT syntax, *subquery_restriction_clause::= on page 16-42*)

object_view_clause::=

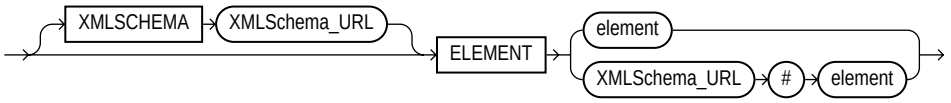


(*inline_constraint::=* on page 7-7 and *out_of_line_constraint::=* on page 7-7—part of *constraints* syntax)

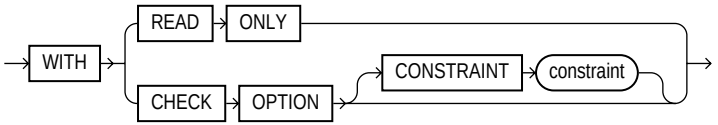
XMLType_view_clause::=



XMLSchema_spec::=



subquery_restriction_clause::=



Keywords and Parameters

OR REPLACE

Specify OR REPLACE to re-create the view if it already exists. You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it.

INSTEAD OF triggers defined in the view are dropped when a view is re-created.

If any materialized views are dependent on *view*, those materialized views will be marked UNUSABLE and will require a full refresh to restore them to a usable state. Invalid materialized views cannot be used by query rewrite and cannot be refreshed until they are recompiled.

See Also:

- ALTER MATERIALIZED VIEW on page 9-92 for information on refreshing invalid materialized views
- *Oracle9i Database Concepts* for information on materialized views in general
- CREATE TRIGGER on page 15-95 for more information about the INSTEAD OF clause

FORCE

Specify FORCE if you want to create the view regardless of whether the view's base tables or the referenced object types exist or the owner of the schema containing the view has privileges on them. These conditions must be true before any SELECT, INSERT, UPDATE, or DELETE statements can be issued against the view.

If the view definition contains any constraints, CREATE VIEW ... FORCE will fail if the base table does not exist or the referenced object type does not exist. CREATE VIEW ... FORCE will also fail if the view definition references a constraint that does not exist.

NO FORCE

Specify NOFORCE if you want to create the view only if the base tables exist and the owner of the schema containing the view has privileges on them. This is the default.

schema

Specify the schema to contain the view. If you omit *schema*, Oracle creates the view in your own schema.

view

Specify the name of the view or the object view.

Restriction on views: If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.

See Also: "Creating a View: Example" on page 16-50

alias

Specify names for the expressions selected by the view's query. The number of aliases must match the number of expressions selected by the view. Aliases must

follow the rules for naming Oracle schema objects. Aliases must be unique within the view.

If you omit the aliases, Oracle derives them from the columns or column aliases in the view's query. For this reason, you **must** use aliases if the view's query contains expressions rather than only column names. Also, you must specify aliases if the view definition includes constraints.

Restriction on view aliases: You cannot specify an alias when creating an object view.

> **See Also:** "Syntax for Schema Objects and Parts in SQL Statements" on page 2-116

inline_constraint and *out_of_line_constraint*

You can specify constraints on views and object views. You define the constraint at the view level using the *out_of_line_constraint* clause. You define the constraint as part of column or attribute specification using the *inline_constraint* clause after the appropriate alias.

Oracle does not enforce view constraints. However, operations on views are subject to the integrity constraints defined on the underlying base tables. This means that you can enforce constraints on views through constraints on base tables.

Restrictions on view constraints: View constraints are a subset of table constraints and are subject to the following restrictions:

- You can specify only unique, primary key, and foreign key constraints on views. However, you can define the view using the `WITH CHECK OPTION` clause, which is equivalent to specifying a check constraint for the view.
- Because view constraints are not enforced directly, you cannot specify `INITIALLY DEFERRED` or `DEFERRABLE`.
- View constraints are supported only in `DISABLE NOVALIDATE` mode. You must specify the keywords `DISABLE NOVALIDATE` when you declare the view constraint, and you cannot specify any other mode.
- You cannot specify the *using_index_clause*, the *exceptions_clause* clause, or the `ON DELETE` clause of the *references_clause*.
- You cannot define view constraints on attributes of an object column.

> **See Also:** *constraints* on page 7-5 for more information on constraints in general and on restrictions on view constraints and on page 16-50"Creating a View with Constraints: Example"

object_view_clause

The *object_view_clause* lets you define a view on an object type.

> **See Also:** "Creating an Object View: Example" on page 16-52

OF *type_name* Clause

Use this clause to explicitly create an **object view** of type *type_name*. The columns of an object view correspond to the top-level attributes of type *type_name*. Each row will contain an object instance and each instance will be associated with an object identifier (OID) as specified in the WITH OBJECT IDENTIFIER clause. If you omit *schema*, Oracle creates the object view in your own schema.

Object tables (as well as XMLType tables, object views, and XMLType views) do not have any column names specified for them. Therefore, Oracle defines a system-generated column SYS_NC_ROWINFO$. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

WITH OBJECT IDENTIFIER Clause

Use the WITH OBJECT IDENTIFIER clause to specify a top-level (root) object view. This clause lets you specify the attributes of the object type that will be used as a key to identify each row in the object view. In most cases these attributes correspond to the primary key columns of the base table. You must ensure that the attribute list is unique and identifies exactly one row in the view.

Restrictions on object views:

- If you try to dereference or pin a primary key REF that resolves to more than one instance in the object view, Oracle returns an error.
- You cannot specify this clause if you are creating a subview, because subviews inherit object identifiers from superviews.

Note: The Oracle8*i*, Release 8.0 syntax WITH OBJECT OID is replaced with this syntax for clarity. The keywords WITH OBJECT OID are supported for backward compatibility, but Oracle Corporation recommends that you use the new syntax WITH OBJECT IDENTIFIER.

If the object view is defined on an object table or an object view, you can omit this clause or specify DEFAULT.

DEFAULT Specify DEFAULT if you want Oracle to use the intrinsic object identifier of the underlying object table or object view to uniquely identify each row.

attribute For *attribute*, specify an attribute of the object type from which Oracle should create the object identifier for the object view.

UNDER Clause

Use the UNDER clause to specify a subview based on an object superview.

To learn whether a view is a superview or a subview, query the SUPERVIEW_NAME column of the USER_, ALL_, or DBA_VIEWS data dictionary views.

Restrictions on subviews:

- You must create a subview in the same schema as the superview.
- The object type *type_name* must be the immediate subtype of *superview*.
- You can create only one subview of a particular type under the same superview.

See Also:

- CREATE TYPE on page 16-3 for information about creating objects
- *Oracle9i Database Reference* for information on data dictionary views

AS *subquery*

Specify a subquery that identifies columns and rows of the table(s) that the view is based on. The select list of the subquery can contain up to 1000 expressions.

If you create views that refer to remote tables and views, the database links you specify must have been created using the CONNECT TO clause of the CREATE DATABASE LINK statement, and you must qualify them with schema name in the view subquery.

If you create a view with the *flashback_clause* in the defining subquery, Oracle does not interpret the AS OF expression at create time but rather each time a user subsequently queries the view.

See Also: "Creating a Join View: Example" on page 16-51

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more information on flashback queries

Restrictions on the view subquery:

- The view subquery cannot select the CURRVAL or NEXTVAL pseudocolumns.
- If the view subquery selects the ROWID, ROWNUM, or LEVEL pseudocolumns, those columns must have aliases in the view subquery.
- If the view subquery uses an asterisk (*) to select all columns of a table, and you later add new columns to the table, the view will not contain those columns until you re-create the view by issuing a CREATE OR REPLACE VIEW statement.
- For object views, the number of elements in the view subquery select list must be the same as the number of top-level attributes for the object type. The datatype of each of the selecting elements must be the same as the corresponding top-level attribute.
- You cannot specify the SAMPLE clause.

The preceding restrictions apply to materialized views as well.

Notes on Creating Updatable Views:

An updatable view is one you can use to insert, update, or delete base table rows. You can create a view to be inherently updatable, or you can create an INSTEAD OF trigger on any view to make it updatable.

To learn whether and in what ways the columns of an inherently updatable view can be modified, query the USER_UPDATABLE_COLUMNS data dictionary view. (The information displayed by this view is meaningful only for inherently updatable views.)

- If you want the view to be inherently updatable, it must not contain any of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list

  - Joins (with some exceptions as described in the paragraphs that follow).
- In addition, if an inherently updatable view contains pseudocolumns or expressions, you cannot update base table rows with an `UPDATE` statement that refers to any of these pseudocolumns or expressions.
- If you want a join view to be updatable, all of the following conditions must be true:
  - The DML statement must affect only one table underlying the join.
  - For an `INSERT` statement, the view must not be created `WITH CHECK OPTION`, and all columns into which values are inserted must come from a **key-preserved table**. A key-preserved table in one for which every primary key or unique key value in the base table is also unique in the join view.
  - For an `UPDATE` statement, all columns updated must be extracted from a key-preserved table. If the view was created `WITH CHECK OPTION`, join columns and columns taken from tables that are referenced more than once in the view must be shielded from `UPDATE`.
- For a `DELETE` statement, if the join results in more than one key-preserved table, then Oracle deletes from the first table named in the `FROM` clause, whether or not the view was created `WITH CHECK OPTION`.

See Also:

- *Oracle9i Database Administrator's Guide* for more information on updatable views
- *Oracle9i Application Developer's Guide - Fundamentals* for more information about updating object views or relational views that support object types
- "Creating an Updatable View: Example" on page 16-50
- "Creating a Join View: Example" on page 16-51 for an example of updatable join views and key-preserved tables
- "Creating an INSTEAD OF Trigger: Example" on page 15-108 for an example of an `INSTEAD OF` trigger on a view that is not inherently updatable

XMLType_view_clause

Use this clause to create an `XMLType` view, which displays data from an XMLSchema-based table of type `XMLType`. The *`XMLSchema_spec`* indicates the

XMLSchema to be used to map the XML data to its object-relational equivalents. The XMLSchema must already have been created before you can create an XMLType view.

Object tables (as well as XMLType tables, object views, and XMLType views) do not have any column names specified for them. Therefore, Oracle defines a system-generated column SYS_NC_ROWINFO$. You can use this column name in queries and to create object views with the WITH OBJECT IDENTIFIER clause.

See Also:

- *Oracle9i XML Database Developer's Guide - Oracle XML DB* for information on XMLType views and XMLSchemas
- "Creating an XMLType View: Example" on page 16-53 and "Creating a View on an XMLType Table: Example" on page 16-53

subquery_restriction_clause

Use the *subquery_restriction_clause* to restrict the subquery in one of the following ways:

WITH READ ONLY Specify WITH READ ONLY if you want no deletes, inserts, or updates to be performed through the view.

See Also: "Creating a Read-Only View: Example" on page 16-52

WITH CHECK OPTION Specify WITH CHECK OPTION to guarantee that inserts and updates performed through the view will result in rows that the view subquery can select. The CHECK OPTION cannot make this guarantee if:

- There is a subquery within the subquery of this view or any view on which this view is based or
- INSERT, UPDATE, or DELETE operations are performed using INSTEAD OF triggers.

CONSTRAINT *constraint* Specify the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

Examples

Creating a View: Example The following statement creates a view of the sample table employees named emp_view. The view shows the employees in department 20 and their annual salary:

```
CREATE VIEW emp_view AS
   SELECT last_name, salary*12 annual_salary
   FROM employees
   WHERE department_id = 20;
```

The view declaration need not define a name for the column based on the expression sal*12, because the subquery uses a column alias (annual_salary) for this expression.

Creating a View with Constraints: Example The following statement creates a restricted view of the sample table hr.employees and defines a unique constraint on the email view column and a primary key constraint for the view on the emp_id view column:

```
CREATE VIEW emp_sal (emp_id, last_name,
      email UNIQUE RELY DISABLE NOVALIDATE,
   CONSTRAINT id_pk PRIMARY KEY (emp_id) RELY DISABLE NOVALIDATE)
   AS SELECT employee_id, last_name, email FROM employees;
```

Creating an Updatable View: Example The following statement creates an updatable view named clerk of all sales and purchasing clerks in the employees table. Only the employees' IDs, last names, department numbers, and jobs are visible in this view, and these columns can be updated only in rows where the employee is a king of clerk:

```
CREATE VIEW clerk AS
   SELECT employee_id, last_name, department_id, job_id
   FROM employees
   WHERE job_id = 'PU_CLERK'
      or job_id = 'SH_CLERK'
      or job_id = 'ST_CLERK';
```

This view lets you change the job_id of a purchasing clerk to purchasing manager (PU_MAN):

```
UPDATE clerk SET job_id = 'PU_MAN' WHERE employee_id = 118;
```

The next example creates the same view WITH CHECK OPTION. You cannot subsequently insert a new row into clerk if the new employee is not a clerk. You

can update an employee's job_id from one type of clerk to another type of clerk, but the update in the preceding statement would fail, because the view cannot access employees with non-clerk job_id.

```
CREATE VIEW clerk AS
   SELECT employee_id, last_name, department_id, job_id
   FROM employees
   WHERE job_id = 'PU_CLERK'
      or job_id = 'SH_CLERK'
      or job_id = 'ST_CLERK'
   WITH CHECK OPTION;
```

Creating a Join View: Example A join view is one whose view subquery contains a join. If at least one column in the join has a unique index, then it may be possible to modify one base table in a join view. You can query USER_UPDATABLE_COLUMNS to see whether the columns in a join view are updatable. For example:

```
CREATE VIEW locations_view AS
   SELECT d.department_id, d.department_name, l.location_id, l.city
   FROM departments d, locations l
   WHERE d.location_id = l.location_id;

SELECT column_name, updatable
   FROM user_updatable_columns
   WHERE table_name = 'LOCATIONS_VIEW';

COLUMN_NAME                    UPD
------------------------------ ---
DEPARTMENT_ID                  YES
DEPARTMENT_NAME                YES
LOCATION_ID                    NO
CITY                           NO
```

In the preceding example, the primary key index on the location_id column of the locations table is not unique in the locations_view view. Therefore, locations is not a key-preserved table and columns from that base table are not updatable.

```
INSERT INTO locations_view VALUES
   (999, 'Entertainment', 87, 'Roma');
INSERT INTO locations_view VALUES
*
ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join
view
```

You can insert, update, or delete a row from the departments base table, because all the columns in the view mapping to the departments table are marked as updatable and because the primary key of departments is retained in the view.

```
INSERT INTO locations_view (department_id, department_name)
   VALUES (999, 'Entertainment');

1 row created.
```

Note: You cannot insert into the table using the view unless the view contains all NOT NULL columns of all tables in the join, unless you have specified DEFAULT values for the NOT NULL columns.

See Also: *Oracle9i Application Developer's Guide - Fundamentals* for more information on updating join views

Creating a Read-Only View: Example The following statement creates a read-only view named customer_ro of the oe.customers table. Only the customers' last names, language, and credit limit are visible in this view:

```
CREATE VIEW customer_ro (name, language, credit)
      AS SELECT cust_last_name, nls_language, credit_limit
      FROM customers
      WITH READ ONLY;
```

Creating an Object View: Example The following example shows the creation of the type inventory_typ in the oc schema, and the oc_inventories view that is based on that type:

```
CREATE TYPE inventory_typ AS OBJECT
    ( product_id          number(6)
    , warehouse           warehouse_typ
    , quantity_on_hand    number(8)
    ) ;

CREATE OR REPLACE VIEW oc_inventories OF inventory_typ
   WITH OBJECT IDENTIFIER (product_id)
   AS SELECT i.product_id,
     warehouse_typ(w.warehouse_id, w.warehouse_name, w.location_id),
     i.quantity_on_hand
   FROM inventories i, warehouses w
   WHERE i.warehouse_id=w.warehouse_id;
```

Creating a View on an XMLType Table: Example The following example builds a regular view on the XMLType table xwarehouses, which was created in "XMLType Table Examples" on page 15-71:

```
CREATE VIEW warehouse_view AS
   SELECT VALUE(p) AS warehouse_xml
   FROM xwarehouses p;
```

You select from such a view as follows:

```
SELECT e.warehouse_xml.getclobval()
   FROM warehouse_view e
   WHERE EXISTSNODE(warehouse_xml, '//Docks') =1;
```

Creating an XMLType View: Example In some cases you may have an object-relational table upon which you would like to build an XMLType view. The following example creates an object-relational table (resembling the XMLType column warehouse_spec in the sample table oe.warehouses), and then creates an XMLType view of that table:

```
CREATE TABLE warehouse_table
(
  WarehouseID       NUMBER,
  Area              NUMBER,
  Docks             NUMBER,
  DockType          VARCHAR2(100),
  WaterAccess       VARCHAR2(10),
  RailAccess        VARCHAR2(10),
  Parking           VARCHAR2(20),
  VClearance        NUMBER
);

INSERT INTO warehouse_table
   VALUES(5, 103000,3,'Side Load','false','true','Lot',15);

CREATE VIEW warehouse_view OF XMLTYPE
 XMLSCHEMA "http://www.oracle.com/xwarehouses.xsd"
    ELEMENT "Warehouse"
    WITH OBJECT ID
    (extract(sys_nc_rowinfo$,
'/Warehouse/Area/text()').getnumberval())
 AS SELECT XMLELEMENT("Warehouse",
            XMLFOREST(WarehouseID as "Building",
                      area as "Area",
                      docks as "Docks",
```

```
                      docktype as "DockType",
                      wateraccess as "WaterAccess",
                      railaccess as "RailAccess",
                      parking as "Parking",
                      VClearance as "VClearance"))
  FROM warehouse_table;
```

You would query this view as follows:

```
SELECT VALUE(e) FROM warehouse_view e;
```

DELETE

Purpose

Use the DELETE statement to remove rows from a table, a partitioned table, a view's base table, or a view's partitioned base table.

Prerequisites

For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

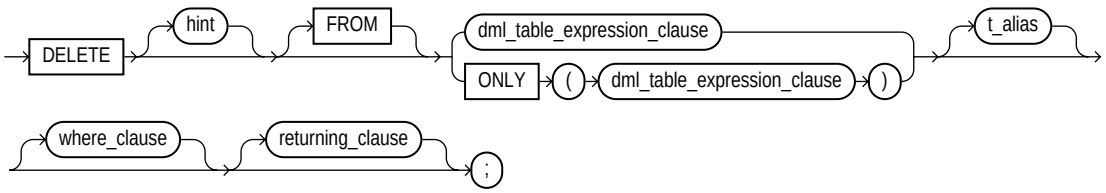
The DELETE ANY TABLE system privilege also allows you to delete rows from any table or table partition, or any view's base table.

You must also have the SELECT privilege on the object from which you want to delete if:

- The object is on a remote database or
- The SQL92_SECURITY initialization parameter is set to TRUE and the DELETE operation references table columns (such as the columns in a *where_clause*).

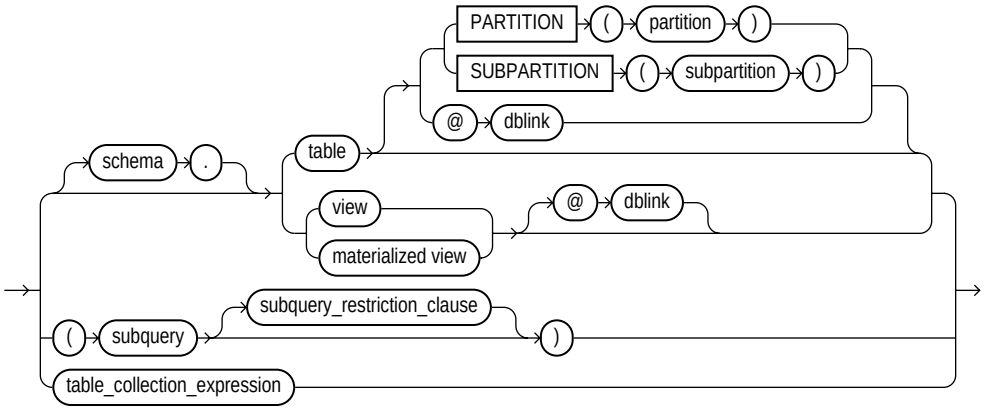
Syntax

delete::=



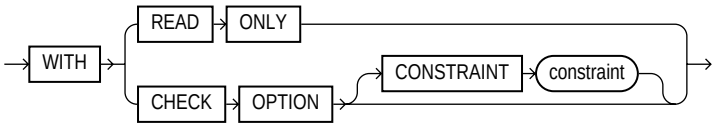
(*DML_table_expression_clause::=* on page 16-56, *where_clause::=* on page 16-56, *returning_clause::=* on page 16-56)

DML_table_expression_clause::=



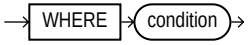
(*subquery::=* on page 18-5, *subquery_restriction_clause::=* on page 16-56, *table_collection_expression::=* on page 16-56)

subquery_restriction_clause::=

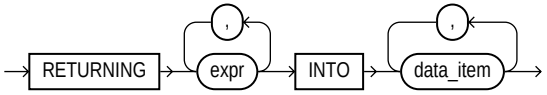


table_collection_expression::=

where_clause::=



returning_clause::=



Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:** "Hints" on page 2-92 and *Oracle9i Database Performance Tuning Guide and Reference* for the syntax and description of hints

from_clause

Use the FROM clause to specify the database objects from which you are deleting rows.

The ONLY syntax is only relevant for views. Use the ONLY clause if the view in the FROM clause belongs to a view hierarchy and you do not want to delete rows from any of its subviews.

DML_table_expression_clause

schema

Specify the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | *view* | *materialized view* | *subquery*

Specify the name of a table or view, or the column or columns resulting from a subquery, from which the rows are to be deleted. If you specify *view*, Oracle deletes rows from the view's base table.

If table (or the base table of view) contains one or more domain index columns, this statements executes the appropriate indextype delete routine.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for more information on these routines

Issuing a DELETE statement against a table fires any DELETE triggers defined on the table.

All table or index space released by the deleted rows is retained by the table and index.

PARTITION (*partition_name*) and SUBPARTITION (*subpartition_name*)

Specify the name of the partition or subpartition within *table* targeted for deletes.

You need not specify the partition name when deleting values from a partitioned table. However, in some cases, specifying the partition name is more efficient than a complicated *where_clause*.

> **See Also:** "Deleting Rows from a Partition: Example" on page 16-63

dblink

Specify the complete or partial name of a database link to a remote database where the table or view is located. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-118 for information on referring to database links and "Deleting Rows from a Remote Database: Example" on page 16-62

If you omit *dblink*, Oracle assumes that the table or view is located on the local database.

subquery_restriction_clause

The *subquery_restriction_clause* lets you restrict the subquery in one of the following ways:

WITH READ ONLY Specify WITH READ ONLY to indicate that the table or view cannot be updated.

WITH CHECK OPTION Specify WITH CHECK OPTION to indicate that Oracle prohibits any changes to the table or view that would produce rows that are not included in the subquery.

CONSTRAINT *constraint* Specify the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

> **See Also:** "Using the WITH CHECK OPTION Clause: Example" on page 18-34

table_collection_expression

The *table_collection_expression* lets you inform Oracle that the value of *collection_expression* should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value (that is, a value whose type is nested table or varray). This process of extracting the elements of a collection is called **collection unnesting**.

Note: In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as "THE subquery". That usage is now deprecated.

You can use a *table_collection_expression* to delete only those rows that also exist in another table.

collection_expression Specify a subquery that selects a nested table column from *table* or *view*.

Note: In earlier releases of Oracle, *table_collection_expression* was expressed as "THE *subquery*". That usage is now deprecated.

Restrictions on the *dml_table_expression_clause*:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- You cannot specify the ORDER BY clause in the subquery of the *dml_table_expression_clause*.
- You cannot delete from a view except through INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function

- A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
- A collection expression in a SELECT list
- A subquery in a SELECT list
- Joins (with some exceptions). See *Oracle9i Database Administrator's Guide* for details.

If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the DELETE statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to `true`.

> **See Also:** ALTER SESSION on page 10-2

table_collection_expression

where_clause

Use the *where_clause* to delete only rows that satisfy the condition. The condition can reference the table and can contain a subquery. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality.

> **See Also:** Chapter 5, "Conditions" for the syntax of *condition*

Note: If this clause contains a *subquery* that refers to remote objects, the DELETE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *dml_table_expression_clause* refers to any remote objects, the UPDATE operation will run serially without notification.

See Also: *parallel_clause* for CREATE TABLE on page 15-56

If you omit *dblink*, Oracle assumes that the table or view is located on the local database.

If you omit the *where_clause*, Oracle deletes all rows of the table or view.

t_alias Provide a **correlation name** for the table, view, subquery, or collection value to be referenced elsewhere in the statement. Table aliases are generally used in DELETE statements with correlated queries.

Note: This alias is required if the *dml_table_expression_clause* references any object type attributes or object type methods.

returning_clause

The returning clause retrieves the rows affected by a DML (INSERT, UPDATE, or DELETE) statement. You can specify this clause for tables and materialized views, and for views with a single base table.

When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and REFs to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the affected rows in bind arrays.

expr Each item in the *expr* list must be a valid expression syntax. All forms are valid except scalar subquery expressions.

INTO The INTO clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.

data_item Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the RETURNING list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the INTO list.

Restrictions:

- You cannot specify the *returning_clause* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

See Also: *PL/SQL User's Guide and Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

Note: This clause lets you return values from deleted columns, and thereby eliminate the need to issue a SELECT statement following the DELETE statement.

See Also: "Using the RETURNING Clause: Example" on page 16-63

Examples

Deleting Rows: Examples The following statement deletes all rows from the sample table oe.product_descriptions:

```
DELETE FROM product_descriptions
   WHERE language_id = 'AR';
```

The following statement deletes from the sample table hr.employees purchasing clerks whose commission rate is less than 10%:

```
DELETE FROM employees
   WHERE job_id = 'PU_CLERK'
   AND commission_pct < .1;
```

The following statement has the same effect as the preceding example, but uses a subquery:

```
DELETE FROM (SELECT * FROM employees)
   WHERE job_id = 'PU_CLERK'
   AND commission_pct < .1;
```

Deleting Rows from a Remote Database: Example The following statement deletes all rows from the accounts table owned by the user blake on a database accessible by the database link dallas:

```
DELETE FROM blake.accounts@dallas;
```

Deleting Nested Table Rows: Example The following example deletes rows of nested table projs where the department number is either 123 or 456, or the department's budget is greater than 456.78:

```
DELETE TABLE(SELECT projs FROM dept d WHERE d.dno = 123) p
   WHERE p.pno IN (123, 456) OR p.budgets > 456.78;
```

Deleting Rows from a Partition: Example The following example removes rows from partition `sales_q1_1998` of the `sh.sales` table:

```
DELETE FROM sales PARTITION (sales_q1_1998)
   WHERE amount_sold > 10000;
```

Using the RETURNING Clause: Example The following example returns column `salary` from the deleted rows and stores the result in bind variable :bnd1. (The bind variable must already have been declared.)

```
DELETE FROM employees
   WHERE job_id = 'SA_REP'
   AND hire_date + TO_YMINTERVAL('01-00') < SYSDATE;
   RETURNING salary INTO :bnd1;
```

DISASSOCIATE STATISTICS

Purpose

Use the `DISASSOCIATE STATISTICS` statement to disassociate a statistics type (or default statistics) from columns, standalone functions, packages, types, domain indexes, or indextypes.

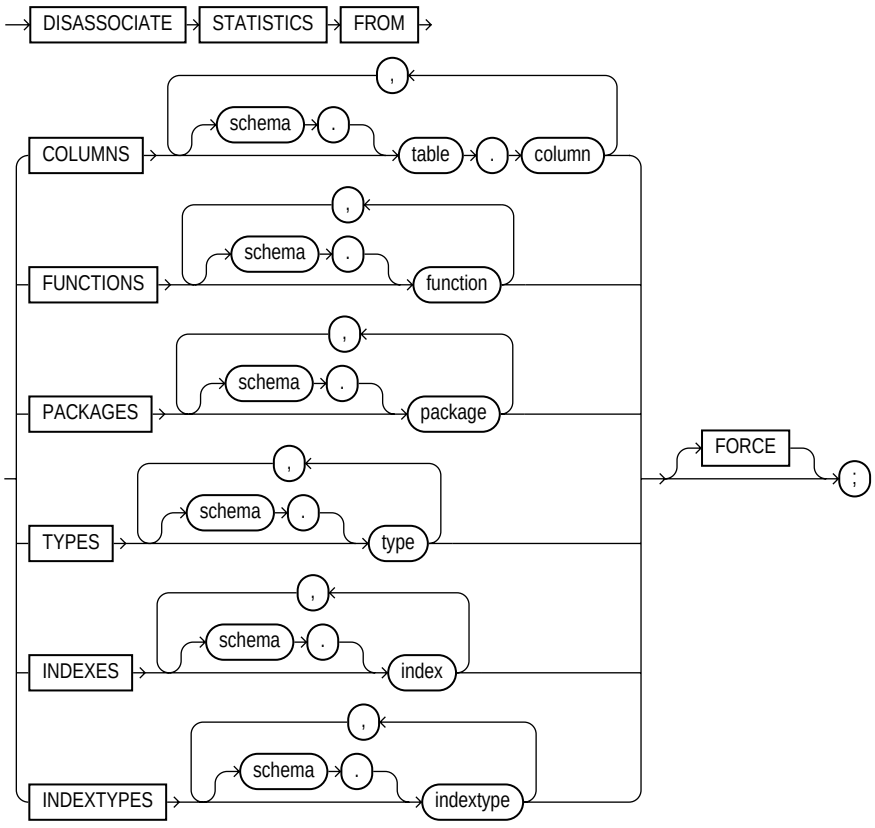
> **See Also:** ASSOCIATE STATISTICS on page 12-48 for more information on statistics type associations

Prerequisites

To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype).

Syntax

disassociate_statistics::=



Keywords and Parameters

FROM COLUMNS | FUNCTIONS | PACKAGES | TYPES | INDEXES | INDEXTYPES

Specify one or more columns, standalone functions, packages, types, domain indexes, or indextypes from which you are disassociating statistics.

If you do not specify *schema*, Oracle assumes the object is in your own schema.

If you have collected user-defined statistics on the object, the statement fails unless you specify FORCE.

FORCE

Specify FORCE to remove the association regardless of whether any statistics exist for the object using the statistics type. If statistics do exist, the statistics are deleted before the association is deleted.

Note: When you drop an object with which a statistics type has been associated, Oracle automatically disassociates the statistics type with the FORCE option and drops all statistics that have been collected with the statistics type.

Example

Disassociating Statistics: Example This statement disassociates statistics from the pack package in the hr schema:

```
DISASSOCIATE STATISTICS FROM PACKAGES oe.emp_mgmt;
```

DROP CLUSTER

Purpose

Use the DROP CLUSTER clause to remove a cluster from the database.

You cannot uncluster an individual table. Instead you must perform these steps:

1. Create a new table with the same structure and contents as the old one, but with no CLUSTER clause.
2. Drop the old table.
3. Use the RENAME statement to give the new table the name of the old one.
4. Grant privileges on the new unclustered table, as grants on the old clustered table do not apply.

> **See Also:** CREATE TABLE on page 15-7, DROP TABLE on page 17-6, RENAME on page 17-87, GRANT on page 17-29 for information on these steps

Prerequisites

The cluster must be in your own schema or you must have the DROP ANY CLUSTER system privilege.

Syntax

drop_cluster::=

Keywords and Parameters

schema

Specify the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.

cluster

Specify the name of the cluster to be dropped. Dropping a cluster also drops the cluster index and returns all cluster space, including data blocks for the index, to the appropriate tablespace(s).

INCLUDING TABLES

Specify `INCLUDING TABLES` to drop all tables that belong to the cluster.

CASCADE CONSTRAINTS

Specify `CASCADE CONSTRAINTS` to drop all referential integrity constraints from tables outside the cluster that refer to primary and unique keys in tables of the cluster. If you omit this clause and such referential integrity constraints exist, Oracle returns an error and does not drop the cluster.

Example

Dropping a Cluster: Examples The following examples drop the clusters created in the "Examples" section of `CREATE CLUSTER` on on page 13-9.

The following statements drops the `language` cluster:

```
DROP CLUSTER language;
```

This statement drops the `personnel` cluster as well as tables `dept_10` and `dept_20` and any referential integrity constraints that refer to primary or unique keys in those tables:

```
DROP CLUSTER personnel
   INCLUDING TABLES
   CASCADE CONSTRAINTS;
```

DROP CONTEXT

Purpose

Use the DROP CONTEXT statement to remove a context namespace from the database.

Note: Removing a context namespace does not invalidate any context under that namespace that has been set for a user session. However, the context will be invalid when the user next attempts to set that context.

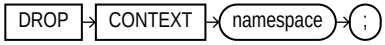
See Also: CREATE CONTEXT on page 13-12 and *Oracle9i Database Concepts* for more information on contexts

Prerequisites

You must have the DROP ANY CONTEXT system privilege.

Syntax

drop_context::=



Keywords and Parameters

namespace

Specify the name of the context namespace to drop. You cannot drop the built-in namespace USERENV.

See Also: SYS_CONTEXT on page 6-153 for information on the USERENV namespace

Example

Dropping an Application Context: Example The following statement drops the context created in CREATE CONTEXT on page 13-12:

```
DROP CONTEXT hr_context;
```

DROP DATABASE LINK

Purpose

Use the DROP DATABASE LINK statement to remove a database link from the database.

> **See Also:** CREATE DATABASE LINK on page 13-35 for information on creating database links

Prerequisites

A private database link must be in your own schema. To drop a PUBLIC database link, you must have the DROP PUBLIC DATABASE LINK system privilege.

Syntax

drop_database_link::=

Keywords and Parameters

PUBLIC

You must specify PUBLIC to drop a PUBLIC database link.

dblink

Specify the name of the database link to be dropped.

Restriction on database links: You cannot drop a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema, because periods are permitted in names of database links. Therefore, Oracle interprets the entire name, such as `ralph.linktosales`, as the name of a database link in your schema rather than as a database link named `linktosales` in the schema `ralph`.

Example

Dropping a Database Link: Example The following statement drops a private database link named `sales.hq.acme.com`:

```
DROP DATABASE LINK sales.hq.acme.com;
```

DROP DIMENSION

Purpose

Use the `DROP DIMENSION` statement to remove the named dimension.

Note: This statement does not invalidate materialized views that use relationships specified in dimensions. However, requests that have been rewritten by query rewrite may be invalidated, and subsequent operations on such views may execute more slowly.

See Also:

- CREATE DIMENSION on page 13-41 for information on creating a dimension
- ALTER DIMENSION on page 9-58 for information on modifying a dimension
- *Oracle9i Database Concepts*

Prerequisites

The dimension must be in your own schema or you must have the `DROP ANY DIMENSION` system privilege to use this statement.

Syntax

drop_dimension::=

Keywords and Parameters

schema

Specify the name of the schema in which the dimension is located. If you omit *schema*, Oracle assumes the dimension is in your own schema.

dimension

Specify the name of the dimension you want to drop. The dimension must already exist.

Example

Dropping a Dimension: Example This example drops the `sh.customers_dim` dimension:

```
DROP DIMENSION customers_dim;
```

> **See Also:** "Creating a Dimension: Example" on page 13-45 and "Modifying a Dimension: Examples" on page 9-60 for examples of creating and modifying this dimension

DROP DIRECTORY

Purpose

Use the `DROP DIRECTORY` statement to remove a directory object from the database.

> **See Also:** CREATE DIRECTORY on page 13-46 for information on creating a directory

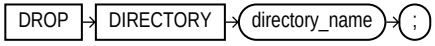
Prerequisites

To drop a directory, you must have the `DROP ANY DIRECTORY` system privilege.

> **Caution:** Do not drop a directory when files in the associated file system are being accessed by PL/SQL or OCI programs.

Syntax

drop_directory::=



Keywords and Parameters

directory_name

Specify the name of the directory database object to be dropped.

Oracle removes the directory object but does not delete the associated operating system directory on the server's file system.

Example

Dropping a Directory: Example The following statement drops the directory object `bfile_dir`:

```
DROP DIRECTORY bfile_dir;
```

> **See Also:** "Creating a Directory: Examples" on page 13-48

DROP FUNCTION

Purpose

Use the DROP FUNCTION statement to remove a standalone stored function from the database.

Note: Do not use this statement to remove a function that is part of a package. Instead, either drop the entire package using the DROP PACKAGE statement or redefine the package without the function using the CREATE PACKAGE statement with the OR REPLACE clause.

See Also:

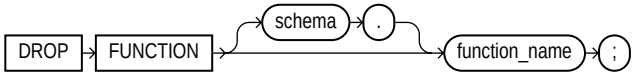
- CREATE FUNCTION on page 13-49 for information on creating a function
- ALTER FUNCTION on page 9-61 for information on modifying a function

Prerequisites

The function must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax

drop_function::=



Keywords and Parameters

schema

Specify the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.

function_name

Specify the name of the function to be dropped.

Oracle invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped function.

If any statistics types are associated with the function, Oracle disassociates the statistics types with the FORCE option and drops any user-defined statistics collected with the statistics type.

See Also:

- *Oracle9i Database Concepts* for more information on how Oracle maintains dependencies among schema objects, including remote objects
- ASSOCIATE STATISTICS on page 12-48 and DISASSOCIATE STATISTICS on page 16-64 for more information on statistics type associations

Example

Dropping a Function: Example The following statement drops the function SecondMax in the sample schema oe and invalidates all objects that depend upon SecondMax:

```
DROP FUNCTION oe.SecondMax;
```

See Also: "Creating a Function: Examples" on page 13-59 for information on creating the SecondMax function

DROP INDEX

Purpose

Use the DROP INDEX statement to remove an index or domain index from the database.

When you drop an index, Oracle invalidates all objects that depend on the underlying table, including views, packages, package bodies, functions, and procedures.

When you drop a global partitioned index, a range-partitioned index, or a hash-partitioned index, all the index partitions are also dropped. If you drop a composite-partitioned index, all the index partitions and subpartitions are also dropped.

In addition, when you drop a domain index:

- Oracle invokes the appropriate routine. For information on these routines, see *Oracle9i Data Cartridge Developer's Guide.*
- If any statistics are associated with the domain index, Oracle disassociates the statistics types with the FORCE clause and removes the user-defined statistics collected with the statistics type.

See Also:

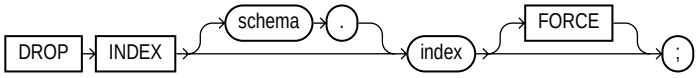
- CREATE INDEX on page 13-62 for information on creating an index
- ALTER INDEX on page 9-64 for information on modifying an index
- The *domain_index_clause* of CREATE INDEX on page 13-62 for more information on domain indexes
- ASSOCIATE STATISTICS on page 12-48 and DISASSOCIATE STATISTICS on page 16-64 for more information on statistics type associations

Prerequisites

The index must be in your own schema or you must have the DROP ANY INDEX system privilege.

Syntax

drop_index::=



Keywords and Parameters

schema

Specify the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index

Specify the name of the index to be dropped. When the index is dropped, all data blocks allocated to the index are returned to the index's tablespace.

Restriction on dropping indexes: You cannot drop a domain index if the index or any of its index partitions is marked `IN_PROGRESS`.

FORCE

`FORCE` applies only to domain indexes. This clause drops the domain index even if the indextype routine invocation returns an error or the index is marked `IN PROGRESS`. Without `FORCE`, you cannot drop a domain index if its indextype routine invocation returns an error or the index is marked `IN PROGRESS`.

Example

Dropping an Index: Example This statement drops an index named `ord_customer_ix_demo` (created in "General Index Examples" on page 13-83):

```
DROP INDEX ord_customer_ix_demo;
```

DROP INDEXTYPE

Purpose

Use the DROP INDEXTYPE statement to drop an indextype, as well as any association with a statistics type.

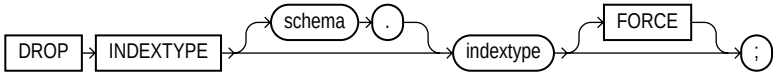
> **See Also:** CREATE INDEXTYPE on page 13-91 for more information on indextypes

Prerequisites

The indextype must be in your own schema or you must have the DROP ANY INDEXTYPE system privilege.

Syntax

drop_indextype::=



Keywords and Parameters

schema

Specify the schema containing the indextype. If you omit *schema*, Oracle assumes the indextype is in your own schema.

indextype

Specify the name of the indextype to be dropped.

If any statistics types have been associated with indextype, Oracle disassociates the statistics type from the indextype and drops any statistics that have been collected using the statistics type.

> **See Also:** ASSOCIATE STATISTICS on page 12-48 and DISASSOCIATE STATISTICS on page 16-64 for more information on statistics associations

FORCE

Specify FORCE to drop the indextype even if the indextype is currently being referenced by one or more domain indexes. Oracle marks those domain indexes INVALID. Without FORCE, you cannot drop an indextype if any domain indexes reference the indextype.

Example

Dropping an Indextype: Example The following statement drops the indextype textindextype and marks INVALID any domain indexes defined on this indextype:

```
DROP INDEXTYPE textindextype FORCE;
```

DROP JAVA

Purpose

Use the DROP JAVA statement to drop a Java source, class, or resource schema object.

See Also:

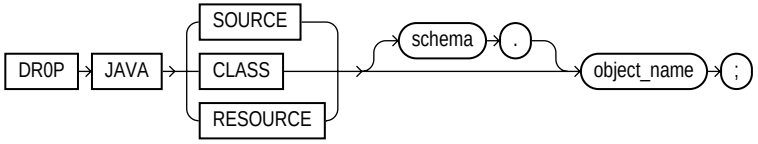
- CREATE JAVA on page 13-94 for information on creating Java objects
- *Oracle9i Java Stored Procedures Developer's Guide* for more information on resolving Java sources, classes, and resources

Prerequisites

The Java source, class, or resource must be in your own schema or you must have the DROP ANY PROCEDURE system privilege. You also must have the EXECUTE object privilege on Java classes to use this command.

Syntax

drop_java::=



Keywords and Parameters

JAVA SOURCE

Specify SOURCE to drop a Java source schema object and all Java class schema objects derived from it.

JAVA CLASS

Specify CLASS to drop a Java class schema object.

JAVA RESOURCE

Specify RESOURCE to drop a Java resource schema object.

object_name

Specify the name of an existing Java class, source, or resource schema object. Enclose the *object_name* in double quotation marks to preserve lower- or mixed-case names.

Example

Dropping a Java Class Object: Example The following statement drops the Java class MyClass:

```
DROP JAVA CLASS "MyClass";
```

DROP LIBRARY

Purpose

Use the DROP LIBRARY statement to remove an external procedure library from the database.

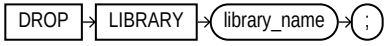
> **See Also:** CREATE LIBRARY on page 14-2 for information on creating a library

Prerequisites

You must have the DROP ANY LIBRARY system privilege.

Syntax

drop_library::=



Keywords and Parameters

library_name

Specify the name of the external procedure library being dropped.

Example

Dropping a Library: Example The following statement drops the ext_lib library (created in "Creating a Library: Examples" on page 14-3):

```
DROP LIBRARY ext_lib;
```

DROP MATERIALIZED VIEW

Purpose

Use the DROP MATERIALIZED VIEW statement to remove an existing materialized view from the database.

Note: The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

See Also:

- CREATE MATERIALIZED VIEW on page 14-5 for more information on materialized views, including a description of the various types of materialized views
- ALTER MATERIALIZED VIEW on page 9-92 for information on modifying a materialized view
- *Oracle9i Replication* for information on materialized views in a replication environment
- *Oracle9i Data Warehousing Guide* for information on materialized views in a data warehousing environment

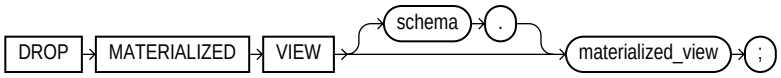
Prerequisites

The materialized view must be in your own schema or you must have the DROP ANY MATERIALIZED VIEW system privilege. You must also have the privileges to drop the internal table, views, and index that Oracle uses to maintain the materialized view's data.

See Also: DROP TABLE on page 17-6, DROP VIEW on page 17-22, and DROP INDEX on page 16-76 for information on privileges required to drop objects that Oracle uses to maintain the materialized view

Syntax

drop_materialized_view::=



Keywords and Parameters

schema

Specify the schema containing the materialized view. If you omit *schema*, Oracle assumes the materialized view is in your own schema.

materialized_view

Specify the name of the existing materialized view to be dropped.

- If you drop a simple materialized view that is the least recently refreshed materialized view of a master table, Oracle automatically purges from the master table's materialized view log only the rows needed to refresh the dropped materialized view.
- If you drop a master table, Oracle does not automatically drop materialized views based on the table. However, Oracle returns an error when it tries to refresh a materialized view based on a master table that has been dropped.
- If you drop a materialized view, any compiled requests that were rewritten to use the materialized view will be invalidated and recompiled automatically. If the materialized view was prebuilt on a table, the table is not dropped, but it can no longer be maintained by the materialized view refresh mechanism.

Examples

Dropping a Materialized View: Examples The following statement drops the materialized view `emp_data` in the sample schema `hr`:

```
DROP MATERIALIZED VIEW emp_data;
```

The following statement drops the `sales_by_month_by_state` materialized view and the underlying table of the materialized view (unless the underlying table was registered in the `CREATE MATERIALIZED VIEW` statement with the `ON PREBUILT TABLE` clause):

```
DROP MATERIALIZED VIEW sales_by_month_by_state;
```

DROP MATERIALIZED VIEW LOG

Purpose

Use the `DROP MATERIALIZED VIEW LOG` statement to remove a materialized view log from the database.

Note: The keyword `SNAPSHOT` is supported in place of `MATERIALIZED VIEW` for backward compatibility.

See Also:

- CREATE MATERIALIZED VIEW on page 14-5 and ALTER MATERIALIZED VIEW on page 9-92 for more information on materialized views
- CREATE MATERIALIZED VIEW LOG on page 14-32 for information on materialized view logs
- *Oracle9i Replication* for information on materialized views in a replication environment
- *Oracle9i Data Warehousing Guide* for information on materialized views in a data warehousing environment

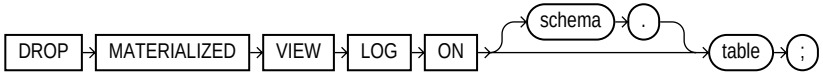
Prerequisites

To drop a materialized view log, you must have the privileges needed to drop a table.

See Also: DROP TABLE on page 17-6

Syntax

drop_materialized_view_log::=



Keywords and Parameters

schema

Specify the schema containing the materialized view log and its master table. If you omit *schema*, Oracle assumes the materialized view log and master table are in your own schema.

table

Specify the name of the master table associated with the materialized view log to be dropped.

After you drop a materialized view log, some materialized views based on the materialized view log's master table can no longer be fast refreshed. These materialized views include rowid materialized views, primary key materialized views, and subquery materialized views.

> **See Also:** *Oracle9i Data Warehousing Guide* for a description of these types of materialized views

Example

Dropping a Materialized View Log: Example The following statement drops the materialized view log on the `oe.customers` master table:

```
DROP MATERIALIZED VIEW LOG ON customers;
```

DROP OPERATOR

Purpose

Use the DROP OPERATOR statement to drop a user-defined operator.

See Also:

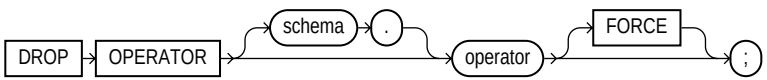
- CREATE OPERATOR on page 14-42 for information on creating operators
- "User-Defined Operators" on page 3-6 and *Oracle9i Data Cartridge Developer's Guide* for more information on operators in general
- ALTER INDEXTYPE on page 9-87 for information on dropping an operator of a user-defined indextype

Prerequisites

The operator must be in your schema or you must have the DROP ANY OPERATOR system privilege.

Syntax

drop_operator::=



Keywords and Parameters

schema

Specify the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.

operator

Specify the name of the operator to be dropped.

FORCE

Specify FORCE to drop the operator even if it is currently being referenced by one or more schema objects (indextypes, packages, functions, procedures, and so on), and marks those dependent objects INVALID. Without FORCE, you cannot drop an operator if any schema objects reference it.

Example

Dropping a User-Defined Operator: Example The following statement drops the operator eq_op:

```
DROP OPERATOR eq_op;
```

Because the FORCE clause is not specified, this operation will fail if any of the bindings of this operator are referenced by an indextype.

DROP OUTLINE

Purpose

Use the DROP OUTLINE statement to drop a stored outline.

See Also:

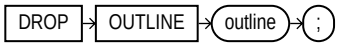
- CREATE OUTLINE on page 14-46 for information on creating an outline
- *Oracle9i Database Performance Tuning Guide and Reference* for more information on outlines in general

Prerequisites

To drop an outline, you must have the DROP ANY OUTLINE system privilege.

Syntax

drop_outline::=



Keywords and Parameters

outline

Specify the name of the outline to be dropped.

After the outline is dropped, if the SQL statement for which the stored outline was created is compiled, the optimizer generates a new execution plan without the influence of the outline.

Example

Dropping an Outline: Example The following statement drops the stored outline called salaries.

```
DROP OUTLINE salaries;
```

DROP PACKAGE

Purpose

Use the DROP PACKAGE statement to remove a stored package from the database. This statement drops the body and specification of a package.

Note: Do not use this statement to remove a single object from a package. Instead, re-create the package without the object using the CREATE PACKAGE and CREATE PACKAGE BODY statements with the OR REPLACE clause.

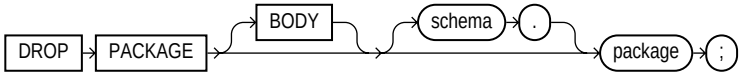
See Also: CREATE PACKAGE on page 14-50

Prerequisites

The package must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax

drop_package::=



Keywords and Parameters

BODY

Specify BODY to drop only the body of the package. If you omit this clause, Oracle drops both the body and specification of the package.

When you drop only the body of a package but not its specification, Oracle does not invalidate dependent objects. However, you cannot call one of the procedures or stored functions declared in the package specification until you re-create the package body.

schema

Specify the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.

package

Specify the name of the package to be dropped.

Oracle invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, Oracle disassociates the statistics types with the FORCE clause and drops any user-defined statistics collected with the statistics types.

See Also:

- *Oracle9i Database Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects
- ASSOCIATE STATISTICS on page 12-48 and DISASSOCIATE STATISTICS on page 16-64

Example

Dropping a Package: Example The following statement drops the specification and body of the emp_mgmt package (created in "Creating a Package: Example" on page 14-53), invalidating all objects that depend on the specification:

```
DROP PACKAGE emp_mgmt;
```

DROP PROCEDURE

Purpose

Use the DROP PROCEDURE statement to remove a standalone stored procedure from the database. Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the DROP PACKAGE statement, or redefine the package without the procedure using the CREATE PACKAGE statement with the OR REPLACE clause.

See Also:

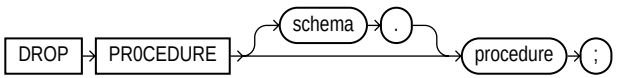
- CREATE PROCEDURE on page 14-62 for information on creating a procedure
- ALTER PROCEDURE on page 9-126 for information on modifying a procedure

Prerequisites

The procedure must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax

drop_procedure::=



Keywords and Parameters

schema

Specify the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.

procedure

Specify the name of the procedure to be dropped.

When you drop a procedure, Oracle invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, Oracle

tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

> **See Also:** *Oracle9i Database Concepts* for information on how Oracle maintains dependencies among schema objects, including remote objects

Example

Dropping a Procedure: Example The following statement drops the procedure `remove_emp` owned by the user `hr` and invalidates all objects that depend upon `remove_emp`:

```
DROP PROCEDURE hr.remove_emp;
```

DROP PROFILE

Purpose

Use the DROP PROFILE statement to remove a profile from the database.

See Also:

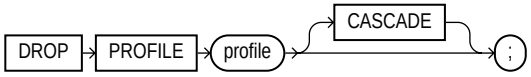
- CREATE PROFILE on page 14-69 on creating a profile
- ALTER PROFILE on page 9-129 on modifying a profile

Prerequisites

You must have the DROP PROFILE system privilege.

Syntax

drop_profile::=



Keywords and Parameters

profile

Specify the name of the profile to be dropped.

Restriction on dropping profiles: You cannot drop the DEFAULT profile.

CASCADE

Specify CASCADE to deassign the profile from any users to whom it is assigned. Oracle automatically assigns the DEFAULT profile to such users. You must specify this clause to drop a profile that is currently assigned to users.

Example

Dropping a Profile: Example The following statement drops the profile app_user (created in "Creating a Profile: Example" on page 14-74):

```
DROP PROFILE app_user CASCADE;
```

Oracle drops the profile `app_user` and assigns the `DEFAULT` profile to any users currently assigned the `app_user` profile.

DROP ROLE

Purpose

Use the DROP ROLE statement to remove a role from the database. When you drop a role, Oracle revokes it from all users and roles to whom it has been granted and removes it from the database. User sessions in which the role is already enabled are not affected. However, no new user session can enable the role after it is dropped.

See Also:

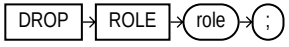
- CREATE ROLE on page 14-77 for information on creating roles
- ALTER ROLE on page 9-136 for information on changing the authorization needed to enable a role
- SET ROLE on page 18-47 for information on disabling roles for the current session

Prerequisites

You must have been granted the role with the ADMIN OPTION or you must have the DROP ANY ROLE system privilege.

Syntax

drop_role::=



Keywords and Parameters

role

Specify the name of the role to be dropped.

Example

Dropping a Role: Example To drop the role dw_manager (created in "Creating a Role: Example" on page 14-79), issue the following statement:

```
DROP ROLE dw_manager;
```

DROP ROLLBACK SEGMENT

Purpose

Use the DROP ROLLBACK SEGMENT to remove a rollback segment from the database. When you drop a rollback segment, all space allocated to the rollback segment returns to the tablespace.

Note: If your database is running in Automatic Undo Management mode, this is the only valid operation on rollback segments. In that mode, you cannot create or alter a rollback segment.

See Also:

- CREATE ROLLBACK SEGMENT on page 14-80 for information on creating a rollback segment
- ALTER ROLLBACK SEGMENT on page 9-138 for information on modifying a rollback segment
- CREATE TABLESPACE on page 15-80

Prerequisites

You must have the DROP ROLLBACK SEGMENT system privilege.

Syntax

drop_rollback_segment::=

Keywords and Parameters

rollback_segment

Specify the name the rollback segment to be dropped.

Restrictions on dropping rollback segments:

- You can drop a rollback segment only if it is offline. To determine whether a rollback segment is offline, query the data dictionary view DBA_ROLLBACK_SEGS. Offline rollback segments have the value AVAILABLE in the STATUS column. You can take a rollback segment offline with the OFFLINE clause of the ALTER ROLLBACK SEGMENT statement.
- You cannot drop the SYSTEM rollback segment.

Example

Dropping a Rollback Segment: Example The following statement drops the rollback segment rbs_ts:

```
DROP ROLLBACK SEGMENT rbs_ts;
```

17

SQL Statements: DROP SEQUENCE to ROLLBACK

This chapter contains the following SQL statements:

- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TABLESPACE
- DROP TRIGGER
- DROP TYPE
- DROP TYPE BODY
- DROP USER
- DROP VIEW
- EXPLAIN PLAN
- GRANT
- INSERT
- LOCK TABLE
- MERGE
- NOAUDIT
- RENAME
- REVOKE
- ROLLBACK

DROP SEQUENCE

Purpose

Use the DROP SEQUENCE statement to remove a sequence from the database.

You can also use this statement to restart a sequence by dropping and then re-creating it. For example, if you have a sequence with a current value of 150 and you would like to restart the sequence with a value of 27, then you can drop the sequence and then re-create it with the same name and a START WITH value of 27.

See Also:

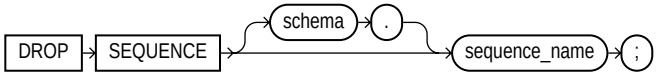
- CREATE SEQUENCE on page 14-87 for information on creating a sequence
- ALTER SEQUENCE on page 9-142 for more information on modifying a sequence

Prerequisites

The sequence must be in your own schema or you must have the DROP ANY SEQUENCE system privilege.

Syntax

drop_sequence::=



Keywords and Parameters

schema

Specify the schema containing the sequence. If you omit *schema*, then Oracle assumes the sequence is in your own schema.

sequence_name

Specify the name of the sequence to be dropped.

Example

Dropping a Sequence: Example The following statement drops the sequence `customers_seq` owned by the user `oe` (created in "Creating a Sequence: Example" on page 14-91). To issue this statement, you must either be connected as user `oe` or have `DROP ANY SEQUENCE` system privilege:

```
DROP SEQUENCE oe.customers_seq;
```

DROP SYNONYM

Purpose

Use the DROP SYNONYM statement to remove a synonym from the database or to change the definition of a synonym by dropping and re-creating it.

> **See Also:** CREATE SYNONYM on page 15-2 for more information on synonyms

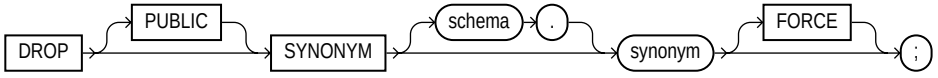
Prerequisites

To drop a private synonym, either the synonym must be in your own schema or you must have the DROP ANY SYNONYM system privilege.

To drop a PUBLIC synonym, you must have the DROP PUBLIC SYNONYM system privilege.

Syntax

drop_synonym::=



Keywords and Parameters

PUBLIC

You must specify PUBLIC to drop a public synonym. You cannot specify *schema* if you have specified PUBLIC.

schema

Specify the schema containing the synonym. If you omit *schema*, then Oracle assumes the synonym is in your own schema.

synonym

Specify the name of the synonym to be dropped.

If you drop a synonym for a materialized view, or its containing table or materialized view, or any of its dependent tables, then Oracle invalidates the materialized view.

Restriction on dropping synonyms: You cannot drop a type synonym that has any dependent tables or user-defined types unless you also specify FORCE.

FORCE

Specify FORCE to drop the synonym even if it has dependent tables or user-defined types.

> **Caution:** Oracle does not recommend that you specify FORCE to drop object synonyms with dependencies. This operation can result in invalidation of other user-defined types or marking UNUSED the table columns that depend on the synonym. For information about type dependencies, see *Oracle9i Application Developer's Guide - Object-Relational Features*.

Example

Dropping a Synonym: Example To drop the public synonym named `customers` (created in "Resolution of Synonyms Example" on page 15-6), issue the following statement:

```
DROP PUBLIC SYNONYM customers;
```

DROP TABLE

Purpose

Use the DROP TABLE statement to remove a table or an object table and all its data from the database.

Caution: You cannot roll back a DROP TABLE statement.

Note: For an external table, this statement removes only the table metadata in the database. It has no affect on the actual data, which resides outside of the database.

Dropping a table invalidates the table's dependent objects and removes object privileges on the table. If you want to re-create the table, then you must regrant object privileges on the table, re-create the table's indexes, integrity constraints, and triggers, and respecify its storage parameters. Truncating has none of these effects. Therefore, removing rows with the TRUNCATE statement can be more efficient than dropping and re-creating a table.

See Also:

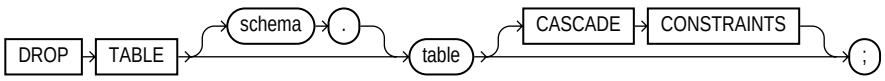
- CREATE TABLE on page 15-7 for information on creating tables
- ALTER TABLE on page 11-2 for information on modifying tables
- TRUNCATE on page 18-54 and DELETE on page 16-55 for information on how to remove data from a table without dropping the table

Prerequisites

The table must be in your own schema or you must have the DROP ANY TABLE system privilege.

Syntax

drop_table::=



Keywords and Parameters

schema

Specify the schema containing the table. If you omit *schema*, then Oracle assumes the table is in your own schema.

table

Specify the name of the table, object table, or index-organized table to be dropped. Oracle automatically performs the following operations:

- Removes all rows from the table.
- Drops all the table's indexes and domain indexes, as well as any triggers defined on the table, regardless of who created them or whose schema contains them. If *table* is partitioned, then any corresponding local index partitions are also dropped.
- Drops all the storage tables of *table*'s nested tables and LOBs.
- If you drop a range-partitioned or hash-partitioned table, then Oracle drops all the table partitions. If you drop a composite-partitioned table, then all the partitions and subpartitions are also dropped.
- For an index-organized table, drops any mapping tables defined on the index-organized table.
- For a domain index, this statement invokes the appropriate drop routines.

  See Also: *Oracle9i Data Cartridge Developer's Guide* for more information on these routines

- If any statistic types are associated with the table, then Oracle disassociates the statistics types with the FORCE clause and removes any user-defined statistics collected with the statistics type.

> **See Also:** ASSOCIATE STATISTICS on page 12-48 and DISASSOCIATE STATISTICS on page 16-64 for more information on statistics type associations

- If the table is not part of a cluster, then Oracle returns all data blocks allocated to the table and its indexes to the tablespaces containing the table and its indexes.

> **Note:** To drop a cluster and all its the tables, use the DROP CLUSTER statement with the INCLUDING TABLES clause to avoid dropping each table individually. See DROP CLUSTER on page 16-67.

- If the table is a base table for a view, a container or master table of a materialized view, or if it is referenced in a stored procedure, function, or package, then Oracle invalidates these dependent objects but does not drop them. You cannot use these objects unless you re-create the table or drop and re-create the objects so that they no longer depend on the table.
- If you choose to re-create the table, then it must contain all the columns selected by the subqueries originally used to define the materialized views and all the columns referenced in the stored procedures, functions, or packages. Any users previously granted object privileges on the views, stored procedures, functions, or packages need not be regranted these privileges.
- If the table is a master table for a materialized view, then the materialized view can still be queried, but it cannot be refreshed unless the table is re-created so that it contains all the columns selected by the materialized view's subquery.
- If the table has a materialized view log, then Oracle drops this log and any other direct-path INSERT refresh information associated with the table.

Restriction on dropping tables: You cannot directly drop the storage table of a nested table. Instead, you must drop the nested table column using the ALTER TABLE ... DROP COLUMN clause.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this clause, and such referential integrity constraints exist, then Oracle returns an error and does not drop the table.

Example

Dropping a Table: Example The following statement drops the `oe.list_customers` table created in "List Partitioning Example" on page 15-73.

```
DROP TABLE list_customers;
```

DROP TABLESPACE

Purpose

Use the DROP TABLESPACE statement to remove a tablespace from the database.

See Also:

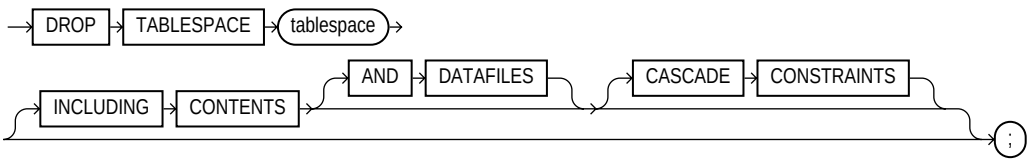
- CREATE TABLESPACE on page 15-80 for information on creating a tablespace
- ALTER TABLESPACE on page 11-101 for information on modifying a tablespace

Prerequisites

You must have the DROP TABLESPACE system privilege. You cannot drop a tablespace if it contains any rollback segments holding active transactions.

Syntax

drop_tablespace::=



Keywords and parameters

tablespace

Specify the name of the tablespace to be dropped.

You can drop a tablespace regardless of whether it is online or offline. Oracle recommends that you take the tablespace offline before dropping it to ensure that no SQL statements in currently running transactions access any of the objects in the tablespace.

You may want to alert any users who have been assigned the tablespace as either a default or temporary tablespace. After the tablespace has been dropped, these users cannot allocate space for objects or sort areas in the tablespace. You can reassign users new default and temporary tablespaces with the ALTER USER statement.

Oracle removes from the data dictionary all metadata about the tablespace and all datafiles and tempfiles in the tablespace. Oracle also automatically drops from the operating system any Oracle-managed datafiles and tempfiles in the tablespace. Other datafiles and tempfiles are not removed from the operating system unless you specify INCLUDING CONTENTS AND DATAFILES.

Restrictions on dropping tablespaces:

- You cannot drop the SYSTEM tablespace.
- You cannot drop a tablespace that contains a domain index or any objects created by a domain index.
- You cannot drop an undo tablespace if it is being used by any instance or if it contains any undo data needed to roll back uncommitted transactions.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* and *Oracle9i Database Concepts* for more information on domain indexes

INCLUDING CONTENTS

Specify INCLUDING CONTENTS to drop all the contents of the tablespace. You must specify this clause to drop a tablespace that contains any database objects. If you omit this clause, and the tablespace is not empty, then Oracle returns an error and does not drop the tablespace.

For partitioned tables, DROP TABLESPACE will fail even if you specify INCLUDING CONTENTS, if the tablespace contains some, but not all:

- Partitions of a range- or hash-partitioned table, or
- Subpartitions of a composite-partitioned table.

> **Note:** If all the partitions of a partitioned table reside in *tablespace*, then DROP TABLESPACE ... INCLUDING CONTENTS will drop *tablespace*, as well as any associated index segments, LOB data segments, and LOB index segments in the other tablespace(s).

For a partitioned index-organized table, if all the primary key index segments are in this tablespace, then this clause will also drop any overflow segments that exist in other tablespaces, as well as any associated mapping table in other tablespaces. If some of the primary key index segments are *not* in this tablespace, then the statement will fail. In that case, before you can drop the tablespace, you must use

ALTER TABLE ... MOVE PARTITION to move those primary key index segments into this tablespace, drop the partitions whose overflow data segments are not in this tablespace, and drop the partitioned index-organized table.

If the tablespace contains a master table of a materialized view, then Oracle invalidates the materialized view.

If the tablespace contains a materialized view log, then Oracle drops this log and any other direct-path INSERT refresh information associated with the table.

AND DATAFILES

When you specify INCLUDING CONTENTS, the AND DATAFILES clause lets you instruct Oracle to delete the associated operating system files as well. Oracle writes a message to the alert log for each operating system file deleted. This clause is not needed for Oracle-managed files.

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints from tables outside *tablespace* that refer to primary and unique keys of tables inside *tablespace*. If you omit this clause and such referential integrity constraints exist, then Oracle returns an error and does not drop the tablespace.

Example

Dropping a Tablespace: Example The following statement drops the tbs_01 tablespace and drops all referential integrity constraints that refer to primary and unique keys inside tbs_01:

```
DROP TABLESPACE tbs_01
    INCLUDING CONTENTS
        CASCADE CONSTRAINTS;
```

Deleting Operating System Files: Example The following example drops the tbs_02 tablespace and deletes all associated operating system datafiles:

```
DROP TABLESPACE tbs_02
   INCLUDING CONTENTS AND DATAFILES;
```

DROP TRIGGER

Purpose

Use the DROP TRIGGER statement to remove a database trigger from the database.

See Also:

- CREATE TRIGGER on page 15-95 for information on creating triggers
- ALTER TRIGGER on page 12-2 for information on enabling, disabling, and compiling triggers

Prerequisites

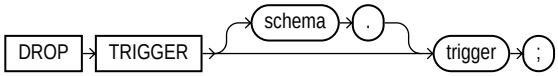
The trigger must be in your own schema or you must have the DROP ANY TRIGGER system privilege.

In addition, to drop a trigger on DATABASE in another user's schema, you must have the ADMINISTER DATABASE TRIGGER system privilege.

See Also: CREATE TRIGGER on page 15-95 for information on these privileges

Syntax

drop_trigger::=



Keywords and Parameters

schema

Specify the schema containing the trigger. If you omit *schema*, then Oracle assumes the trigger is in your own schema.

trigger

Specify the name of the trigger to be dropped. Oracle removes it from the database and does not fire it again.

Example

Dropping a Trigger: Example The following statement drops the `order` trigger in the schema `oe`:

```
DROP TRIGGER hr.salary_check;
```

DROP TYPE

Purpose

Use the DROP TYPE statement to drop the specification and body of an object type, a varray, or nested table type.

See Also:

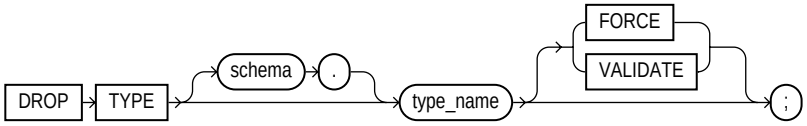
- DROP TYPE BODY on page 17-18 for information on dropping just the body of an object type
- CREATE TYPE on page 16-3 for information on creating types

Prerequisites

The object type, varray, or nested table type must be in your own schema or you must have the DROP ANY TYPE system privilege.

Syntax

drop_type::=



Keywords and Parameters

schema

Specify the schema containing the type. If you omit *schema*, then Oracle assumes the type is in your own schema.

type_name

Specify the name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If *type_name* is a supertype, then this statement will fail unless you also specify FORCE. If you specify FORCE, then Oracle invalidates all subtypes depending on this supertype.

If *type_name* is a statistics type, then this statement will fail unless you also specify FORCE. If you specify FORCE, then Oracle first disassociates all objects that are associated with *type_name*, and then drops *type_name*.

> **See Also:** ASSOCIATE STATISTICS on page 12-48 and DISASSOCIATE STATISTICS on page 16-64 for more information on statistics types

If *type_name* is an object type that has been associated with a statistics type, then Oracle first attempts to disassociate *type_name* from the statistics type and then drop *type_name*. However, if statistics have been collected using the statistics type, then Oracle will be unable to disassociate *type_name* from the statistics type, and this statement will fail.

If *type_name* is an implementation type for an indextype, then the indextype will be marked INVALID.

If *type_name* has a public synonym defined on it, then Oracle will also drop the synonym.

Unless you specify FORCE, you can drop only object types, nested tables, or varray types that are standalone schema objects with no dependencies. This is the default behavior.

> **See Also:** CREATE INDEXTYPE on page 13-91

FORCE

Specify FORCE to drop the type even if it has dependent database objects. Oracle marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible.

Caution: Oracle does not recommend that you specify FORCE to drop types with dependencies. This operation is not recoverable and could cause the data in the dependent tables or columns to become inaccessible. For information about type dependencies, see *Oracle9i Application Developer's Guide - Fundamentals*.

VALIDATE

If you specify VALIDATE when dropping a type, then Oracle checks for stored instances of this type within substitutable columns of any of its supertypes. If no such instances are found, then Oracle completes the drop operation.

This clause is meaningful only for subtypes. Oracle Corporation recommends the use of this option to safely drop subtypes that do not have any "explicit" type or table dependencies.

Example

Dropping an Object Type: Example The following statement removes object type `person_t` (created in "Type Hierarchy Example" on page 16-22):

```
DROP TYPE person_t;
```

DROP TYPE BODY

Purpose

Use the DROP TYPE BODY statement to drop the body of an object type, varray, or nested table type. When you drop a type body, the object type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the object type, although you cannot call the member functions.

See Also:

- DROP TYPE on page 17-15 for information on dropping the specification of an object along with the body
- CREATE TYPE BODY on page 16-25 for more information on type bodies

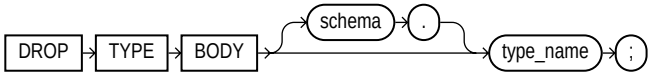
Prerequisites

The object type body must be in your own schema, and you must have

- The CREATE TYPE or CREATE ANY TYPE system privilege, or
- The DROP ANY TYPE system privilege

Syntax

drop_type_body::=



Keywords and Parameters

schema

Specify the schema containing the object type. If you omit *schema*, then Oracle assumes the object type is in your own schema.

type_name

Specify the name of the object type body to be dropped.

Restriction on dropping type bodies: You can drop a type body only if it has no type or table dependencies.

Example

Dropping an Object Type Body: Example The following statement removes object type body `data_typ` (created in "Updating a Type Body: Example" on page 16-30):

```
DROP TYPE BODY data_typ;
```

DROP USER

Purpose

Use the DROP USER statement to remove a database user and optionally remove the user's objects.

See Also:

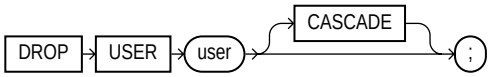
- CREATE USER on page 16-32 for information on creating a user
- ALTER USER on page 12-21 for information on modifying the definition of a user

Prerequisites

You must have the DROP USER system privilege.

Syntax

drop_user::=



Keywords and Parameters

user

Specify the user to be dropped. Oracle does not drop users whose schemas contain objects unless you specify CASCADE or unless you first explicitly drop the user's objects.

CASCADE

Specify CASCADE to drop all objects in the user's schema before dropping the user. You must specify this clause to drop a user whose schema contains any objects.

- If the user's schema contains tables, then Oracle drops the tables and automatically drops any referential integrity constraints on tables in other schemas that refer to primary and unique keys on these tables.

- If this clause results in tables being dropped, then Oracle also drops all domain indexes created on columns of those tables and invokes appropriate drop routines.

  See Also: *Oracle9i Data Cartridge Developer's Guide* for more information on these routines

- Oracle invalidates, but does not drop, the following objects in *other* schemas: views or synonyms for objects in the dropped user's schema; and stored procedures, functions, or packages that query objects in the dropped user's schema.
- Oracle does not drop materialized views in other schemas that are based on tables in the dropped user's schema. However, because the base tables no longer exist, the materialized views in the other schemas can no longer be refreshed.
- Oracle drops all triggers in the user's schema.
- Oracle does *not* drop roles created by the user.

Caution: Oracle also drops with FORCE all types owned by the user. See the FORCE keyword of DROP TYPE on page 17-16.

Examples

Dropping a Database User: Example If user Sidney's schema contains no objects, then you can drop `sidney` by issuing the statement:

```
DROP USER sidney;
```

If Sidney's schema contains objects, then you must use the CASCADE clause to drop `sidney` and the objects:

```
DROP USER sidney CASCADE;
```

DROP VIEW

Purpose

Use the DROP VIEW statement to remove a view or an object view from the database. You can change the definition of a view by dropping and re-creating it.

See Also:

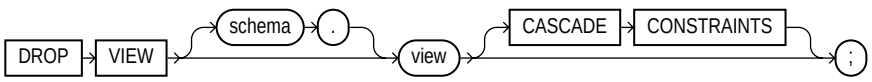
- CREATE VIEW on page 16-39 for information on creating a view
- ALTER VIEW on page 12-30 for information on modifying a view

Prerequisites

The view must be in your own schema or you must have the DROP ANY VIEW system privilege.

Syntax

drop_view::=



Keywords and Parameters

schema

Specify the schema containing the view. If you omit *schema*, then Oracle assumes the view is in your own schema.

view

Specify the name of the view to be dropped.

Oracle does not drop views, materialized views, and synonyms that refer to the view but marks them INVALID. You can drop them or redefine views and synonyms, or you can define other views in such a way that the invalid views and synonyms become valid again.

If any subviews have been defined on view, then Oracle invalidates the subviews as well. To learn if the view has any subviews, query the SUPERVIEW_NAME column of the USER_, ALL_, or DBA_VIEWS data dictionary views.

See Also:

- CREATE TABLE on page 15-7 and CREATE SYNONYM on page 15-2
- ALTER MATERIALIZED VIEW on page 9-92 for information on revalidating invalid materialized views

CASCADE CONSTRAINTS

Specify CASCADE CONSTRAINTS to drop all referential integrity constraints that refer to primary and unique keys in the view to be dropped. If you omit this clause, and such constraints exist, then the DROP statement will fail.

Example

Dropping a View: Example The following statement drops the emp_view view "Creating a View: Example" on page 16-50):

```
DROP VIEW emp_view;
```

EXPLAIN PLAN

Purpose

Use the EXPLAIN PLAN statement to determine the execution plan Oracle follows to execute a specified SQL statement. This statement inserts a row describing each step of the execution plan into a specified table. You can also issue the EXPLAIN PLAN statement as part of the SQL trace facility.

If you are using cost-based optimization, then this statement also determines the cost of executing the statement. If any domain indexes are defined on the table, then user-defined CPU and I/O costs will also be inserted.

The definition of a sample output table PLAN_TABLE is available in a SQL script on your distribution media. Your output table must have the same column names and datatypes as this table. The common name of this script is UTLXPLAN.SQL. The exact name and location depend on your operating system.

Note: Oracle provides information on cached cursors through several dynamic performance views:

- For information on the work areas used by SQL cursors, query V$SQL_WORKAREA.
- For information on the execution plan for a cached cursor, query V$SQL_PLAN.
- For execution statistics at each step or operation of an execution plan of cached cursors (for example, number of produced rows, number of blocks read), query V$SQL_PLAN_STATISTICS view.
- For a selective precomputed join of the preceding three views, query V$SQL_PLAN_STATISTICS_ALL.

See Also:

- *Oracle9i Database Performance Tuning Guide and Reference* for information on the output of EXPLAIN PLAN, how to use the SQL trace facility, and how to generate and interpret execution plans
- *Oracle9i Database Reference* for information on dynamic performance views

Prerequisites

To issue an EXPLAIN PLAN statement, you must have the privileges necessary to insert rows into an existing output table that you specify to hold the execution plan.

You must also have the privileges necessary to execute the SQL statement for which you are determining the execution plan. If the SQL statement accesses a view, then you must have privileges to access any tables and views on which the view is based. If the view is based on another view that is based on a table, then you must have privileges to access both the other view and its underlying table.

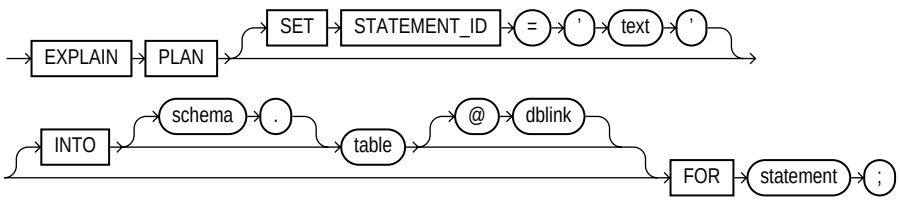
To examine the execution plan produced by an EXPLAIN PLAN statement, you must have the privileges necessary to query the output table.

The EXPLAIN PLAN statement is a data manipulation language (DML) statement, rather than a data definition language (DDL) statement. Therefore, Oracle does not implicitly commit the changes made by an EXPLAIN PLAN statement. If you want to keep the rows generated by an EXPLAIN PLAN statement in the output table, then you must commit the transaction containing the statement.

See Also: INSERT on page 17-54 and SELECT on page 18-4 for information on the privileges you need to populate and query the plan table

Syntax

explain_plan::=



Keywords and Parameters

SET STATEMENT_ID Clause

Specify the value of the STATEMENT_ID column for the rows of the execution plan in the output table. You can then use this value to identify these rows among others in the output table. Be sure to specify a STATEMENT_ID value if your output table contains rows from many execution plans. If you omit this clause, then the STATEMENT_ID value defaults to null.

INTO *table* Clause

Specify the name of the output table, and optionally its schema and database. This table must exist before you use the EXPLAIN PLAN statement.

If you omit *schema*, then Oracle assumes the table is in your own schema.

The *dblink* can be a complete or partial name of a database link to a remote Oracle database where the output table is located. You can specify a remote output table only if you are using Oracle's distributed functionality. If you omit *dblink*, then Oracle assumes the table is on your local database.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-118 for information on referring to database links

If you omit INTO altogether, then Oracle assumes an output table named PLAN_TABLE in your own schema on your local database.

FOR *statement* Clause

Specify a SELECT, INSERT, UPDATE, DELETE, CREATE TABLE, CREATE INDEX, or ALTER INDEX ... REBUILD statement for which the execution plan is generated.

Notes:

- If *statement* includes the *parallel_clause*, then the resulting execution plan will indicate parallel execution. However, EXPLAIN PLAN actually inserts the statement into the plan table, so that the parallel DML statement you submit is no longer the first DML statement in the transaction. This violates the Oracle restriction of one parallel DML statement in a single transaction, and the statement will be executed serially. To maintain parallel execution of the statements, you must commit or roll back the EXPLAIN PLAN statement, and then submit the parallel DML statement.
- To determine the execution plan for an operation on a temporary table, EXPLAIN PLAN must be run from the same session, because the data in temporary tables is session specific.

Examples

EXPLAIN PLAN Examples The following statement determines the execution plan and cost for an UPDATE statement and inserts rows describing the execution plan into the specified plan_table table with the STATEMENT_ID value of 'Raise in Tokyo':

```
EXPLAIN PLAN
    SET STATEMENT_ID = 'Raise in Tokyo'
    INTO plan_table
    FOR UPDATE employees
        SET salary = salary * 1.10
        WHERE department_id =
           (SELECT department_id FROM departments
               WHERE location_id = 1200);
```

The following SELECT statement queries the plan_table table and returns the execution plan and the cost:

```
SELECT LPAD(' ',2*(LEVEL-1))||operation operation, options,
object_name, position
    FROM plan_table
    START WITH id = 0 AND statement_id = 'Raise in Tokyo'
    CONNECT BY PRIOR id = parent_id AND
    statement_id = 'Raise in Tokyo';
```

The query returns this execution plan:

```
OPERATION           OPTIONS         OBJECT_NAME          POSITION
------------------- --------------- ----------------- ----------
UPDATE STATEMENT
  UPDATE                            EMPLOYEES
    INDEX           RANGE SCAN      EMP_DEPARTMENT_IX
      TABLE ACCESS  BY INDEX ROWID  DEPARTMENTS
        INDEX       RANGE SCAN      DEPT_LOCATION_IX
```

The value in the POSITION column of the first row shows that the statement has a cost of 1.

EXPLAIN PLAN: Partitioned Example The sample table sh.sales is partitioned on the time_id column. Partition sales_q3_2000 contains time values less than Oct. 1, 2000, and there is a local index sales_time_bix on the time_id column.

Consider the query:

```
EXPLAIN PLAN FOR
   SELECT * FROM sales
      WHERE time_id BETWEEN :h AND '01-OCT-2000';
```

where :h represents an already declared bind variable. EXPLAIN PLAN executes this query with PLAN_TABLE as the output table. The basic execution plan, including partitioning information, is obtained with the following query:

```
SELECT operation, options, partition_start, partition_stop,
   partition_id FROM plan_table;
```

GRANT

Purpose

Use the GRANT statement to grant:

- System privileges to users and roles
- Roles to users and roles. Both privileges and roles are either local, global, or external. Table 17–1 lists the system privileges (organized by the database object operated upon). Table 17–2 lists Oracle predefined roles.
- Object privileges for a particular object to users, roles, and PUBLIC. Table 17–3 summarizes the object privileges that you can grant on each type of object. Table 17–4 lists object privileges and the operations that they authorize.

Note: You can authorize database users to use roles through means other than the database and the GRANT statement. For example, some operating systems have facilities that let you grant roles to Oracle users with the initialization parameter OS_ROLES. If you choose to grant roles to users through operating system facilities, then you cannot also grant roles to users with the GRANT statement, although you can use the GRANT statement to grant system privileges to users and system privileges and roles to other roles.

See Also:

- CREATE USER on page 16-32 and CREATE ROLE on page 14-77 for definitions of local, global, and external privileges
- *Oracle9i Database Administrator's Guide* for information about other authorization methods
- REVOKE on page 17-89 for information on revoking grants

Prerequisites

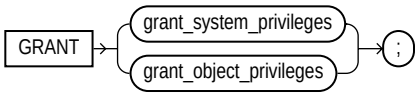
To grant a system privilege, you must either have been granted the system privilege with the ADMIN OPTION or have been granted the GRANT ANY PRIVILEGE system privilege.

To grant a role, you must either have been granted the role with the ADMIN OPTION or have been granted the GRANT ANY ROLE system privilege, or you must have created the role.

To grant an object privilege, you must own the object or the owner of the object must have granted you the object privileges with the GRANT OPTION. This rule applies to users with the DBA role.

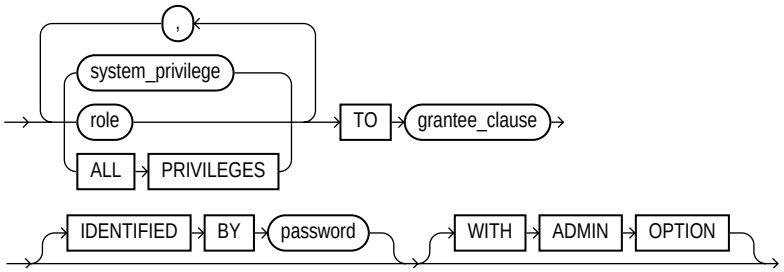
Syntax

grant::=



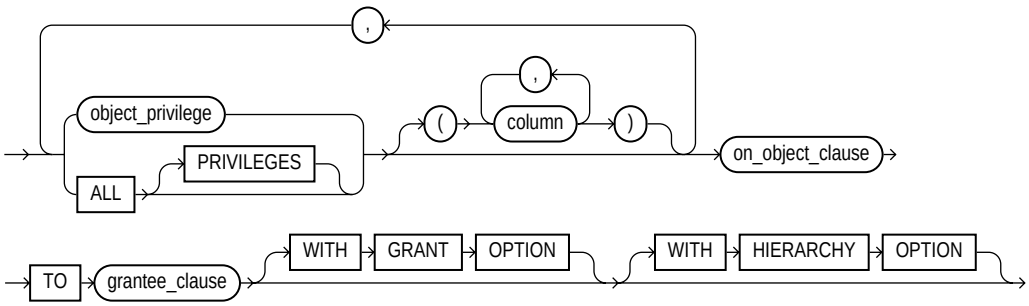
(*grant_system_privileges::=* on page 17-30, *grant_object_privileges::=* on page 17-30)

grant_system_privileges::=



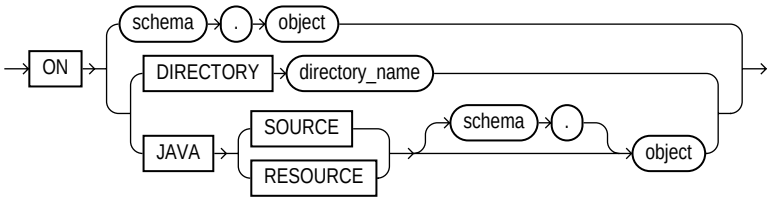
(*grantee_clause::=* on page 17-31)

grant_object_privileges::=

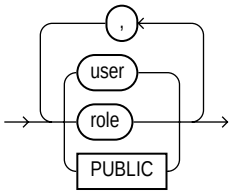


(*on_object_clause::=* on page 17-31, *grantee_clause::=* on page 17-31)

on_object_clause::=



grantee_clause::=



Keywords and Parameters

grant_system_privileges

system_privilege

Specify the system privilege you want to grant. Table 17–1 lists the system privileges (organized by the database object operated upon).

- If you grant a privilege to a **user**, then Oracle adds the privilege to the user's privilege domain. The user can immediately exercise the privilege.

  See Also: Granting a System Privilege to a User: Example on page 17-50

- If you grant a privilege to a **role**, then Oracle adds the privilege to the role's privilege domain. Users who have been granted and have enabled the role can immediately exercise the privilege. Other users who have been granted the role can enable the role and exercise the privilege.

  See Also: "Granting System Privileges to a Role: Example" on page 17-51

- If you grant a privilege to **PUBLIC**, then Oracle adds the privilege to the privilege domains of each user. All users can immediately perform operations authorized by the privilege.

Oracle provides a shortcut for specifying all system privileges at once:

- `ALL PRIVILEGES`: Specify `ALL PRIVILEGES` to grant all the system privileges listed in Table 17–1, " System Privileges" on page 17-36, except the `SELECT ANY DICTIONARY` privilege.

role

Specify the role you want to grant. You can grant an Oracle predefined role or a user-defined role. Table 17–2 lists the predefined roles.

- If you grant a role to a **user**, then Oracle makes the role available to the user. The user can immediately enable the role and exercise the privileges in the role's privilege domain.
- If you grant a role to another **role**, then Oracle adds the granted role's privilege domain to the grantee role's privilege domain. Users who have been granted the grantee role can enable it and exercise the privileges in the granted role's privilege domain.

  See Also: "Granting a Role to a Role: Example" on page 17-51

- If you grant a role to **PUBLIC**, then Oracle makes the role available to all users. All users can immediately enable the role and exercise the privileges in the roles privilege domain.

  See Also: `CREATE ROLE` on page 14-77 for information on creating a user-defined role

IDENTIFIED BY Clause

Use the `IDENTIFIED BY` clause to specifically identify an existing user by password or to create a nonexistent user. This clause is not valid if the grantee is a role or `PUBLIC`. If the user specified in the *`grantee_clause`* does not exist, then Oracle creates the user with the password and with the privileges and roles specified in this clause.

See Also: CREATE USER on page 16-32 for restrictions on usernames and passwords

WITH ADMIN OPTION

Specify WITH ADMIN OPTION to enable the grantee to:

- Grant the role to another user or role, unless the role is a GLOBAL role
- Revoke the role from another user or role
- Alter the role to change the authorization needed to access it
- Drop the role

If you grant a system privilege or role to a user without specifying WITH ADMIN OPTION, and then subsequently grant the privilege or role to the user WITH ADMIN OPTION, then the user has the ADMIN OPTION on the privilege or role.

To revoke the ADMIN OPTION on a system privilege or role from a user, you must revoke the privilege or role from the user altogether and then grant the privilege or role to the user without the ADMIN OPTION.

> **See Also:** "Granting a Role with the Admin Option: Example" on page 17-51

grantee_clause

TO *grantee_clause* identifies users or roles to which the system privilege, role, or object privilege is granted.

Restriction on grantees: A user, role, or PUBLIC cannot appear more than once in TO *grantee_clause.*

PUBLIC Specify PUBLIC to grant the privileges to all users.

Restrictions on granting system privileges and roles:

- A privilege or role cannot appear more than once in the list of privileges and roles to be granted.
- You cannot grant a role to itself.
- You cannot grant a role IDENTIFIED GLOBALLY to anything.
- You cannot grant a role IDENTIFIED EXTERNALLY to a global user or global role.
- You cannot grant roles circularly. For example, if you grant the role banker to the role teller, then you cannot subsequently grant teller to banker.

grant_object_privileges

object_privilege

Specify the object privilege you want to grant. You can specify any of the values shown in Table 17–3. See also Table 17–4.

Restriction on object privileges: A privilege cannot appear more than once in the list of privileges to be granted.

ALL [PRIVILEGES]

Specify ALL to grant all the privileges for the object that you have been granted with the GRANT OPTION. The user who owns the schema containing an object automatically has all privileges on the object with the GRANT OPTION. (The keyword PRIVILEGES is provided for semantic clarity and is optional.)

column

Specify the table or view column on which privileges are to be granted. You can specify columns only when granting the INSERT, REFERENCES, or UPDATE privilege. If you do not list columns, then the grantee has the specified privilege on all columns in the table or view.

For information on existing column object grants, query the USER_,ALL_, and DBA_COL_PRIVS data dictionary view.

> **See Also:** *Oracle9i Database Reference* for information on the data dictionary views and "Granting Multiple Object Privileges on Individual Columns: Example" on page 17-53

on_object_clause

The *on_object_clause* identifies the object on which the privileges are granted. Directory schema objects and Java source and resource schema objects are identified separately because they reside in separate namespaces.

If you can make this grant only because you have the GRANT ANY OBJECT PRIVILEGE system privilege—that is, you are not the owner of *object*, nor do you have *object_privilege* on *object* WITH GRANT OPTION—then the effect of this grant is that you are acting on behalf of the object owner. The *_TAB_PRIVS data dictionary views will reflect that this grant was made by the owner of *object*.

See Also:

- "Granting Object Privileges to a Role" Example" on page 17-51
- "Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example" on page 17-99 for more information on using the GRANT ANY OBJECT PRIVILEGE system privilege for revoke operations

WITH GRANT OPTION

Specify WITH GRANT OPTION to enable the grantee to grant the object privileges to other users and roles.

Restriction on WITH GRANT OPTION: You can specify WITH GRANT OPTION only when granting to a user or to PUBLIC, not when granting to a role.

WITH HIERARCHY OPTION

Specify WITH HIERARCHY OPTION to grant the specified object privilege on all subobjects of *object*, including subobjects created subsequent to this statement (such as subviews created under a view).

Note: This clause is meaningful only in combination with the SELECT object privilege.

---

object Specify the schema object on which the privileges are to be granted. If you do not qualify *object* with *schema*, then Oracle assumes the object is in your own schema. The object can be one of the following types:

- Table, view, or materialized view
- Sequence
- Procedure, function, or package
- User-defined type
- Synonym for any of the preceding items
- Directory, library, operator, or indextype
- Java source, class, or resource

Note: You cannot grant privileges directly to a single partition of a partitioned table.

See Also: "Granting Object Privileges on a Table to a User: Example" on page 17-52, "Granting Object Privileges on a View: Example" on page 17-52, and "Granting Object Privileges to a Sequence in Another Schema: Example" on page 17-52

DIRECTORY *directory_name* Specify a directory schema object on which privileges are to be granted. You cannot qualify *directory_name* with a schema name.

See Also: CREATE DIRECTORY on page 13-46 and "Granting an Object Privilege on a Directory: Example" on page 17-52

JAVA SOURCE | RESOURCE The JAVA clause lets you specify a Java source or resource schema object on which privileges are to be granted.

See Also: CREATE JAVA on page 13-94

Listings of System and Object Privileges

Table 17–1 System Privileges

System Privilege Name	Operations Authorized
CLUSTERS:	
CREATE CLUSTER	Create clusters in grantee's schema
CREATE ANY CLUSTER	Create a cluster in any schema. Behaves similarly to CREATE ANY TABLE.
ALTER ANY CLUSTER	Alter clusters in any schema
DROP ANY CLUSTER	Drop clusters in any schema
CONTEXTS:	
CREATE ANY CONTEXT	Create any context namespace

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
DROP ANY CONTEXT	Drop any context namespace
DATABASE:	
ALTER DATABASE	Alter the database
ALTER SYSTEM	Issue ALTER SYSTEM statements
AUDIT SYSTEM	Issue AUDIT *sql_statements* statements
DATABASE LINKS:	
CREATE DATABASE LINK	Create private database links in grantee's schema
CREATE PUBLIC DATABASE LINK	Create public database links
DROP PUBLIC DATABASE LINK	Drop public database links
DEBUGGING:	
DEBUG CONNECT SESSION	Connect the current session to a debugger that uses the Java Debug Wire Protocol (JDWP).
DEBUG ANY PROCEDURE	Debug all PL/SQL and Java code in any database object; display information on all SQL statements executed by the application **Note:** Granting this privilege is equivalent to granting the DEBUG object privilege on all applicable objects in the database.
DIMENSIONS:	
CREATE DIMENSION	Create dimensions in the grantee's schema
CREATE ANY DIMENSION	Create dimensions in any schema
ALTER ANY DIMENSION	Alter dimensions in any schema
DROP ANY DIMENSION	Drop dimensions in any schema
DIRECTORIES	
CREATE ANY DIRECTORY	Create directory database objects
DROP ANY DIRECTORY	Drop directory database objects
INDEXTYPES:	
CREATE INDEXTYPE	Create an indextype in the grantee's schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
CREATE ANY INDEXTYPE	Create an indextype in any schema
ALTER ANY INDEXTYPE	Modify indextypes in any schema
DROP ANY INDEXTYPE	Drop an indextype in any schema
EXECUTE ANY INDEXTYPE	Reference an indextype in any schema
INDEXES:	
CREATE ANY INDEX	Create in any schema a domain index or an index on any table in any schema
ALTER ANY INDEX	Alter indexes in any schema
DROP ANY INDEX	Drop indexes in any schema
QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables or views in any schema
LIBRARIES:	
CREATE LIBRARY	Create external procedure / function libraries in grantee's schema
CREATE ANY LIBRARY	Create external procedure / function libraries in any schema
DROP ANY LIBRARY	Drop external procedure / function libraries in any schema
MATERIALIZED VIEWS:	
CREATE MATERIALIZED VIEW	Create a materialized view in the grantee's schema
CREATE ANY MATERIALIZED VIEW	Create materialized views in any schema
ALTER ANY MATERIALIZED VIEW	Alter materialized views in any schema
DROP ANY MATERIALIZED VIEW	Drop materialized views in any schema
QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables or views in any schema
ON COMMIT REFRESH	Create a refresh-on-commit materialized view on any table in the database Alter a refresh-on-demand materialized on any table in the database to refresh-on-commit
FLASHBACK ANY TABLE	Issue a SQL flashback query on any table, view, or materialized view in any schema. (This privilege is not needed to execute the DBMS_FLASHBACK procedures.)
OPERATORS:	
CREATE OPERATOR	Create an operator and its bindings in the grantee's schema
CREATE ANY OPERATOR	Create an operator and its bindings in any schema
DROP ANY OPERATOR	Drop an operator in any schema
EXECUTE ANY OPERATOR	Reference an operator in any schema
OUTLINES:	
CREATE ANY OUTLINE	Create public outlines that can be used in any schema that uses outlines
ALTER ANY OUTLINE	Modify outlines
DROP ANY OUTLINE	Drop outlines
SELECT ANY OUTLINE	Create a clone private outline from a public outline
PROCEDURES:	
CREATE PROCEDURE	Create stored procedures, functions, and packages in grantee's schema
CREATE ANY PROCEDURE	Create stored procedures, functions, and packages in any schema
ALTER ANY PROCEDURE	Alter stored procedures, functions, or packages in any schema
DROP ANY PROCEDURE	Drop stored procedures, functions, or packages in any schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
EXECUTE ANY PROCEDURE	Execute procedures or functions (standalone or packaged) Reference public package variables in any schema
PROFILES:	
CREATE PROFILE	Create profiles
ALTER PROFILE	Alter profiles
DROP PROFILE	Drop profiles
ROLES:	
CREATE ROLE	Create roles
ALTER ANY ROLE	Alter any role in the database
DROP ANY ROLE	Drop roles
GRANT ANY ROLE	Grant any role in the database
ROLLBACK SEGMENTS:	
CREATE ROLLBACK SEGMENT	Create rollback segments
ALTER ROLLBACK SEGMENT	Alter rollback segments
DROP ROLLBACK SEGMENT	Drop rollback segments
SEQUENCES:	
CREATE SEQUENCE	Create sequences in grantee's schema
CREATE ANY SEQUENCE	Create sequences in any schema
ALTER ANY SEQUENCE	Alter any sequence in the database
DROP ANY SEQUENCE	Drop sequences in any schema
SELECT ANY SEQUENCE	Reference sequences in any schema
SESSIONS:	
CREATE SESSION	Connect to the database
ALTER RESOURCE COST	Set costs for session resources

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
ALTER SESSION	Issue ALTER SESSION statements
RESTRICTED SESSION	Logon after the instance is started using the SQL*Plus STARTUP RESTRICT statement
SNAPSHOTS: See MATERIALIZED VIEWS	
SYNONYMS:	
CREATE SYNONYM	Create synonyms in grantee's schema
CREATE ANY SYNONYM	Create private synonyms in any schema
CREATE PUBLIC SYNONYM	Create public synonyms
DROP ANY SYNONYM	Drop private synonyms in any schema
DROP PUBLIC SYNONYM	Drop public synonyms
TABLES: **Note:** For external tables, the only valid privileges are CREATE ANY TABLE, ALTER ANY TABLE, DROP ANY TABLE, and SELECT ANY TABLE.	
CREATE TABLE	Create tables in grantee's schema
CREATE ANY TABLE	Create tables in any schema. The owner of the schema containing the table must have space quota on the tablespace to contain the table.
ALTER ANY TABLE	Alter any table or view in any schema
BACKUP ANY TABLE	Use the Export utility to incrementally export objects from the schema of other users
DELETE ANY TABLE	Delete rows from tables, table partitions, or views in any schema
DROP ANY TABLE	Drop or truncate tables or table partitions in any schema
INSERT ANY TABLE	Insert rows into tables and views in any schema
LOCK ANY TABLE	Lock tables and views in any schema
SELECT ANY TABLE	Query tables, views, or materialized views in any schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
FLASHBACK ANY TABLE	Issue a SQL flashback query on any table, view, or materialized view in any schema. (This privilege is not needed to execute the DBMS_FLASHBACK procedures.)
UPDATE ANY TABLE	Update rows in tables and views in any schema
TABLESPACES:	
CREATE TABLESPACE	Create tablespaces
ALTER TABLESPACE	Alter tablespaces
DROP TABLESPACE	Drop tablespaces
MANAGE TABLESPACE	Take tablespaces offline and online and begin and end tablespace backups
UNLIMITED TABLESPACE	Use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, then the user's schema objects remain but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
TRIGGERS:	
CREATE TRIGGER	Create a database trigger in grantee's schema
CREATE ANY TRIGGER	Create database triggers in any schema
ALTER ANY TRIGGER	Enable, disable, or compile database triggers in any schema
DROP ANY TRIGGER	Drop database triggers in any schema
ADMINISTER DATABASE TRIGGER	Create a trigger on DATABASE. (You must also have the CREATE TRIGGER or CREATE ANY TRIGGER privilege.)
TYPES:	
CREATE TYPE	Create object types and object type bodies in grantee's schema
CREATE ANY TYPE	Create object types and object type bodies in any schema
ALTER ANY TYPE	Alter object types in any schema
DROP ANY TYPE	Drop object types and object type bodies in any schema

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
EXECUTE ANY TYPE	Use and reference object types and collection types in any schema, and invoke methods of an object type in any schema if you make the grant to a specific user. If you grant EXECUTE ANY TYPE to a role, then users holding the enabled role will not be able to invoke methods of an object type in any schema.
UNDER ANY TYPE	Create subtypes under any nonfinal object types.
USERS:	
CREATE USER	Create users. This privilege also allows the creator to: ■ Assign quotas on any tablespace ■ Set default and temporary tablespaces ■ Assign a profile as part of a CREATE USER statement
ALTER USER	Alter any user. This privilege authorizes the grantee to: ■ Change another user's password or authentication method ■ Assign quotas on any tablespace ■ Set default and temporary tablespaces ■ Assign a profile and default roles
BECOME USER	Become another user. (Required by any user performing a full database import.)
DROP USER	Drop users
VIEWS:	
CREATE VIEW	Create views in grantee's schema
CREATE ANY VIEW	Create views in any schema
DROP ANY VIEW	Drop views in any schema
UNDER ANY VIEW	Create subviews under any object views
FLASHBACK ANY TABLE	Issue a SQL flashback query on any table, view, or materialized view in any schema. (This privilege is not needed to execute the DBMS_FLASHBACK procedures.)

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
MISCELLANEOUS:	
ANALYZE ANY	Analyze any table, cluster, or index in any schema
AUDIT ANY	Audit any object in any schema using AUDIT *schema_objects* statements
COMMENT ANY TABLE	Comment on any table, view, or column in any schema
EXEMPT ACCESS POLICY	Bypass fine-grained access control **Caution:** This is a very powerful system privilege, as it lets the grantee bypass application-driven security policies. Database administrators should use caution when granting this privilege.
FORCE ANY TRANSACTION	Force the commit or rollback of any in-doubt distributed transaction in the local database Induce the failure of a distributed transaction
FORCE TRANSACTION	Force the commit or rollback of grantee's in-doubt distributed transactions in the local database
GRANT ANY OBJECT PRIVILEGE	Grant any object privilege Revoke any object privilege that was granted by the object owner or by some other user with the GRANT ANY OBJECT PRIVILEGE privilege
GRANT ANY PRIVILEGE	Grant any system privilege
RESUMABLE	Enable resumable space allocation

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–1 (Cont.) System Privileges

System Privilege Name	Operations Authorized
SELECT ANY DICTIONARY	Query any data dictionary object in the SYS schema. This privilege lets you selectively override the default FALSE setting of the O7_DICTIONARY_ACCESSIBILITY initialization parameter.
SYSDBA	Perform STARTUP and SHUTDOWN operations
	ALTER DATABASE: open, mount, back up, or change character set
	CREATE DATABASE
	ARCHIVELOG and RECOVERY
	CREATE SPFILE
	Includes the RESTRICTED SESSION privilege
SYSOPER	Perform STARTUP and SHUTDOWN operations
	ALTER DATABASE OPEN | MOUNT | BACKUP
	ARCHIVELOG and RECOVERY
	CREATE SPFILE
	Includes the RESTRICTED SESSION privilege

Note: When you grant a privilege on "ANY" object (for example, CREATE ANY CLUSTER), you give the user access to that type of object in all schemas, including the SYS schema. If you want to prohibit access to objects in the SYS schema, set the initialization parameter O7_DICTIONARY_ACCESSIBILITY to FALSE. Then privileges granted on "ANY" object will allow access to any schema except SYS.

Table 17–2 Oracle Predefined Roles

Predefined Role	Purpose
CONNECT, RESOURCE, and DBA	These roles are provided for compatibility with previous versions of Oracle. You can determine the privileges encompassed by these roles by querying the DBA_SYS_PRIVS data dictionary view. **Note:** Oracle Corporation recommends that you design your own roles for database security rather than relying on these roles. These roles may not be created automatically by future versions of Oracle. **See Also:** *Oracle9i Database Reference* for a description of this view
DELETE_CATALOG_ROLE EXECUTE_CATALOG_ROLE SELECT_CATALOG_ROLE	These roles are provided for accessing data dictionary views and packages. **See Also:** *Oracle9i Database Administrator's Guide* for more information on these roles
EXP_FULL_DATABASE IMP_FULL_DATABASE	These roles are provided for convenience in using the Import and Export utilities. **See Also:** *Oracle9i Database Utilities* for more information on these roles
AQ_USER_ROLE AQ_ADMINISTRATOR_ROLE	You need these roles to use Oracle's Advanced Queuing functionality. **See Also:** *Oracle9i Application Developer's Guide - Advanced Queuing* for more information on these roles
SNMPAGENT	This role is used by Enterprise Manager / Intelligent Agent. **See Also:** *Oracle Enterprise Manager Administrator's Guide*
RECOVERY_CATALOG_OWNER	You need this role to create a user who owns a recovery catalog. **See Also:** *Oracle9i User-Managed Backup and Recovery Guide* for more information on recovery catalogs
HS_ADMIN_ROLE	A DBA using Oracle's heterogeneous services feature needs this role to access appropriate tables in the data dictionary. **See Also:** *Oracle9i Heterogeneous Connectivity Administrator's Guide* and *Oracle9i Supplied PL/SQL Packages and Types Reference* for more information

Table 17–3 Object Privileges Available for Particular Objects

Object Privilege	Table	View	Sequence	Procedures, Functions, Packages[a]	Materialized View	Directory	Library	User-defined Type	Operator	Indextype
ALTER	X	—	X	—	—	—	—	—	—	—
DELETE	X	X	—	—	X[b]	—	—	—	—	—
EXECUTE	—	—	—	X	—	—	X	X	X	X
DEBUG	X	X	—	X	—	—	—	X	—	—
FLASHBACK	X	X	—	—	X	—	—	—	—	—
INDEX	X	—	—	—	—	—	—	—	—	—
INSERT	X	X	—	—	X[b]	—	—	—	—	—
ON COMMIT REFRESH	X	—	—	—	—	—	—	—	—	—
QUERY REWRITE	X	—	—	—	—	—	—	—	—	—
READ	—	—	—	—	—	X	—	—	—	—
REFERENCES	X	X	—	—	—	—	—	—	—	—
SELECT	X	X	X	—	X	—	—	—	—	—
UNDER	—	X	—	—	—	—	—	X	—	—
UPDATE	X	X	—	—	X[b]	—	—	—	—	—
WRITE	—	—	—	—	—	X	—	—	—	—

[a]Oracle treats a Java class, source, or resource as if it were a procedure for purposes of granting object privileges.
[b]The DELETE, INSERT, and UPDATE privileges can be granted only to updatable materialized views.

Table 17–4 Object Privileges and the Operations They Authorize

Object Privilege	Operations Authorized
The following **table privileges** authorize operations on a table. Any one of following object privileges allows the grantee to lock the table in any lock mode with the LOCK TABLE statement. **Note:** For external tables, the only valid object privileges are ALTER and SELECT.	
ALTER	Change the table definition with the ALTER TABLE statement.
DELETE	Remove rows from the table with the DELETE statement. **Note:** You must grant the SELECT privilege on the table along with the DELETE privilege if the table is on a remote database.
DEBUG	Access, through a debugger: ■ PL/SQL code in the body of any triggers defined on the table ■ Information on SQL statements that reference the table directly
INDEX	Create an index on the table with the CREATE INDEX statement.
INSERT	Add new rows to the table with the INSERT statement.
REFERENCES	Create a constraint that refers to the table. You cannot grant this privilege to a role.
SELECT	Query the table with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement. **Note:** You must grant the SELECT privilege on the table along with the UPDATE privilege if the table is on a remote database.
The following **view privileges** authorize operations on a view. Any one of the following object privileges allows the grantee to lock the view in any lock mode with the LOCK TABLE statement. To grant a privilege on a view, you must have that privilege with the GRANT OPTION on all of the view's base tables.	
DEBUG	Access, through a debugger: ■ PL/SQL code in the body of any triggers defined on the view ■ Information on SQL statements that reference the view directly
DELETE	Remove rows from the view with the DELETE statement.
INSERT	Add new rows to the view with the INSERT statement.
REFERENCES	Define foreign key constraints on the view.
SELECT	Query the view with the SELECT statement.

Table 17–4 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Operations Authorized
UNDER	Create a subview under this view. You can grant this object privilege only if you have the UNDER ANY VIEW privilege WITH GRANT OPTION on the immediate superview of this view.
UPDATE	Change data in the view with the UPDATE statement.
The following **sequence privileges** authorize operations on a sequence.	
ALTER	Change the sequence definition with the ALTER SEQUENCE statement.
SELECT	Examine and increment values of the sequence with the CURRVAL and NEXTVAL pseudocolumns.
The following **procedure, function, and package privilege** authorizes operations on procedures, functions, and packages. This privilege also applies to **Java sources, classes, and resources**, which Oracle treats as though they were procedures for purposes of granting object privileges.	
DEBUG	Access, through a debugger, all public and nonpublic variables, methods, and types defined on the procedure, function, or package. Place a breakpoint or stop at a line or instruction boundary within the procedure, function, or package. This privilege grants access to the declarations in the method or package specification and body.
EXECUTE	Compile the procedure or function or execute it directly, or access any program object declared in the specification of a package. Access, through a debugger, public variables, types, and methods defined on the procedure, function, or package. This privilege grants access to the declarations in the method or package specification only. **Note:** Users do not need this privilege to execute a procedure, function, or package indirectly. **See Also:** *Oracle9i Database Concepts* and *Oracle9i Application Developer's Guide - Fundamentals*
The following **materialized view privileges** authorize operations on a materialized view.	
ON COMMIT REFRESH	Create a refresh-on-commit materialized on the specified table.
QUERY REWRITE	Create a materialized view for query rewrite using the specified table.
SELECT	Query the materialized view with the SELECT statement.
Synonym privileges are the same as the privileges for the base object. Granting a privilege on a synonym is equivalent to granting the privilege on the base object. Similarly, granting a privilege on a base object is equivalent to granting the privilege on all synonyms for the object. If you grant to a user a privilege on a synonym, then the user can use either the synonym name or the base object name in the SQL statement that exercises the privilege.	

Table 17–4 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Operations Authorized
The following **directory privileges** provide secured access to the files stored in the operating system directory to which the directory object serves as a pointer. The directory object contains the full path name of the operating system directory where the files reside. Because the files are actually stored outside the database, Oracle server processes also need to have appropriate file permissions on the file system server. Granting object privileges on the directory database object to individual database users, rather than on the operating system, allows Oracle to enforce security during file operations.	
READ	Read files in the directory.
WRITE	Write files in the directory. This privilege is useful only in connection with external tables. It allows the grantee to determine whether the external table agent can write a log file, or a bad file to the directory. **Restriction**: This privilege does not allow the grantee to write to a BFILE.
The following **library privileges** authorize operations on a library	
EXECUTE	Use and reference the specified object and to invoke its methods.
The following **object type privilege** authorizes operations on a database object type	
DEBUG	Access, through a debugger, all public and nonpublic variables, methods, and types defined on the object type. Place a breakpoint or stop at a line or instruction boundary within the type body.
EXECUTE	Use and reference the specified object and to invoke its methods. Access, through a debugger, public variables, types, and methods defined on the object type.
UNDER	Create a subtype under this type. You can grant this object privilege only if you have the UNDER ANY TYPE privilege WITH GRANT OPTION on the immediate supertype of this type.
The following **indextype privilege** authorizes operations on indextypes.	
EXECUTE	Reference an indextype.
The following **operator privilege** authorizes operations on user-defined operators.	
EXECUTE	Reference an operator.

Examples

Granting a System Privilege to a User: Example To grant the CREATE SESSION system privilege to the sample user hr, allowing hr to log on to Oracle, issue the following statement:

```
GRANT CREATE SESSION
   TO hr;
```

Granting System Privileges to a Role: Example To grant appropriate system privileges to a data warehouse manager role (which was created in the "Creating a Role: Example" on page 14-79) :

```
GRANT
     CREATE ANY MATERIALIZED VIEW
   , ALTER ANY MATERIALIZED VIEW
   , DROP ANY MATERIALIZED VIEW
   , QUERY REWRITE
   , GLOBAL QUERY REWRITE
   TO dw_manager
   WITH ADMIN OPTION;
```

`dw_manager`'s privilege domain now contains the system privileges related to materialized views.

Granting a Role with the Admin Option: Example To grant the `dw_manager` role with the `ADMIN OPTION` to the sample user `sh`, issue the following statement:

```
GRANT dw_manager
   TO sh
   WITH ADMIN OPTION;
```

User `sh` can now perform the following operations with the `dw_manager` role:

- Enable the role and exercise any privileges in the role's privilege domain, including the `CREATE MATERIALIZED VIEW` system privilege
- Grant and revoke the role to and from other users
- Drop the role

Granting Object Privileges to a Role" Example To grant the `SELECT` object privileges to a data warehouse user role (which was created in the "Creating a Role: Example" on page 14-79) :

```
GRANT SELECT ON sh.sales TO warehouse_user;
```

Granting a Role to a Role: Example The following statement grants the `warehouse_user` role to the `dw_manager` role (both roles were created in the "Creating a Role: Example" on page 14-79):

```
GRANT warehouse_user TO dw_manager;
```

The dw_manager role now contains all of the privileges in the domain of the warehouse_user role.

Granting an Object Privilege on a Directory: Example To grant READ on directory bfile_dir to user hr, with the GRANT OPTION, issue the following statement:

```
GRANT READ ON DIRECTORY bfile_dir TO hr
   WITH GRANT OPTION;
```

Granting Object Privileges on a Table to a User: Example To grant all privileges on the table oe.bonuses (created in "Merging into a Table: Example" on page 17-80) to the user hr with the GRANT OPTION, issue the following statement:

```
GRANT ALL ON bonuses TO hr
   WITH GRANT OPTION;
```

hr can subsequently perform the following operations:

- Exercise any privilege on the bonuses table
- Grant any privilege on the bonuses table to another user or role

Granting Object Privileges on a View: Example To grant SELECT and UPDATE privileges on the view emp_view (created in "Creating a View: Example" on page 16-50) to all users, issue the following statement:

```
GRANT SELECT, UPDATE
   ON emp_view TO PUBLIC;
```

All users can subsequently query and update the view of employee details.

Granting Object Privileges to a Sequence in Another Schema: Example To grant SELECT privilege on the orders_seq sequence in the schema oe to the user hr, issue the following statement:

```
GRANT SELECT
   ON oe.orders_seq TO hr;
```

hr can subsequently generate the next value of the sequence with the following statement:

```
SELECT oe.orders_seq.NEXTVAL
   FROM DUAL;
```

Granting Multiple Object Privileges on Individual Columns: Example To grant to user oe the REFERENCES privilege on the employee_id column and the UPDATE privilege on the employee_id, salary, and commission_pct columns of the employees table in the schema hr, issue the following statement:

```
GRANT REFERENCES (employee_id),
      UPDATE (employee_id, salary, commission_pct)
   ON hr.employees
   TO oe;
```

oe can subsequently update values of the employee_id, salary, and commission_pct columns. oe can also define referential integrity constraints that refer to the employee_id column. However, because the GRANT statement lists only these columns, oe cannot perform operations on any of the other columns of the employees table.

For example, oe can create a table with a constraint:

```
CREATE TABLE dependent
   (dependno   NUMBER,
    dependname VARCHAR2(10),
    employee   NUMBER
   CONSTRAINT in_emp REFERENCES hr.employees(employee_id) );
```

The constraint in_emp ensures that all dependents in the dependent table correspond to an employee in the employees table in the schema hr.

INSERT

Purpose

Use the INSERT statement to add rows to a table, a view's base table, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or an object view's base table.

Prerequisites

For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, then you must have INSERT privilege on the view.

If you have the INSERT ANY TABLE system privilege, then you can also insert rows into any table or any view's base table.

Conventional and Direct-Path INSERT

You can use the INSERT statement to insert data into a table, partition, or view in two ways: conventional INSERT and direct-path INSERT. When you issue a conventional INSERT statement, Oracle reuses free space in the table into which you are inserting and maintains referential integrity constraints. With direct-path INSERT, Oracle appends the inserted data after existing data in the table. Data is written directly into datafiles, bypassing the buffer cache. Free space in the existing data is not reused. This alternative enhances performance during insert operations and is similar to the functionality of Oracle's direct-path loader utility, SQL*Loader.

Direct-path INSERT is subject to a number of restrictions. If any of these restrictions is violated, then Oracle executes conventional INSERT serially without returning any message (unless otherwise noted):

- You can have multiple direct-path INSERT statements in a single transaction, with or without other DML statements. However, after one DML statement alters a particular table, partition, or index, no other DML statement in the transaction can access that table, partition, or index.
- Queries that access the same table, partition, or index are allowed before the direct-path INSERT statement, but not after it.

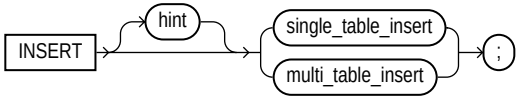
- If any serial or parallel statement attempts to access a table that has already been modified by a direct-path `INSERT` in the same transaction, then Oracle returns an error and rejects the statement.
- The `ROW_LOCKING` initialization parameter cannot be set to `INTENT`.
- The target table cannot be index organized or clustered.
- The target table cannot contain object type or LOB columns.
- The target table cannot have any triggers or referential integrity constraints defined on it.
- The target table cannot be replicated.
- A transaction containing a direct-path `INSERT` statement cannot be or become distributed.

See Also:

- *Oracle9i Database Concepts* for a more complete description of direct-path `INSERT`
- *Oracle9i Database Utilities* for information on SQL*Loader
- *Oracle9i Database Performance Tuning Guide and Reference* for information on how to tune parallel direct-path `INSERT`

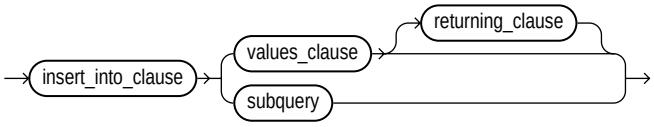
Syntax

insert::=



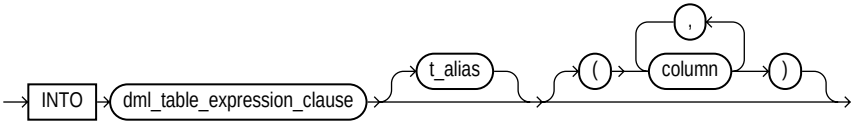
(*single_table_insert::=* on page 17-55, *multi_table_insert::=* on page 17-56)

single_table_insert::=



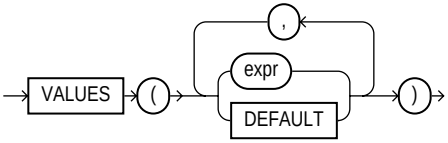
(*insert_into_clause::=* on page 17-56, *values_clause::=* on page 17-56, *returning_clause::=* on page 17-56, *subquery::=* on page 18-5)

insert_into_clause::=

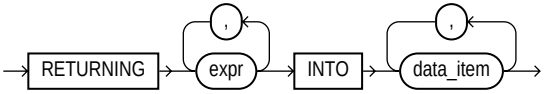


(*DML_table_expression_clause::=* on page 17-57)

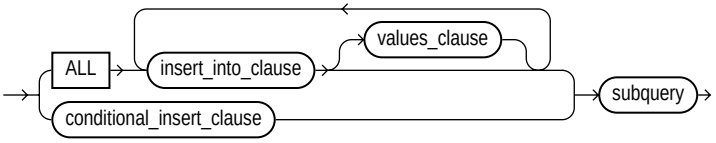
values_clause::=



returning_clause::=

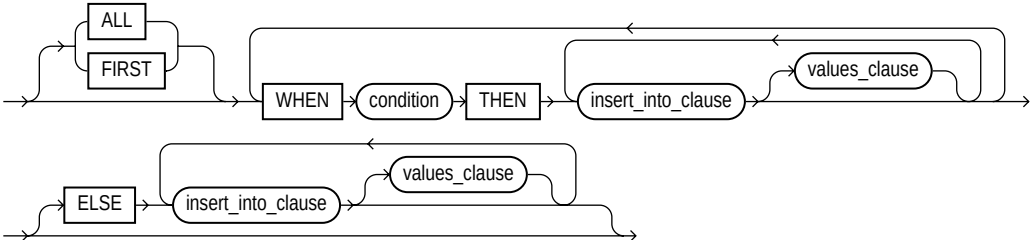


multi_table_insert::=



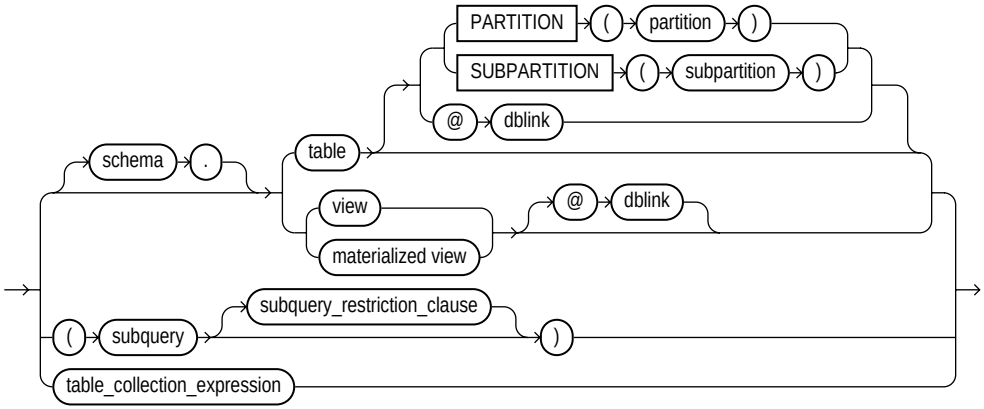
(*insert_into_clause::=* on page 17-56, *values_clause::=* on page 17-56, *conditional_insert_clause::=* on page 17-56, *subquery::=* on page 18-5)

conditional_insert_clause::=



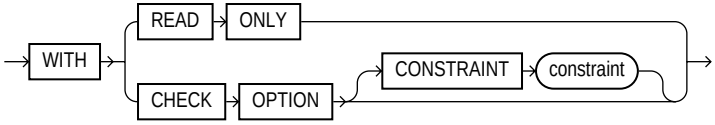
(*insert_into_clause::=* on page 17-56, *values_clause::=* on page 17-56)

DML_table_expression_clause::=



(*subquery::=* on page 18-5—part of SELECT syntax, *subquery_restriction_clause::=* on page 17-57, *table_collection_expression::=* on page 17-57)

subquery_restriction_clause::=



table_collection_expression::=

Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

For a multitable insert, if you specify the PARALLEL hint for any target table, then the entire multitable insert statement is parallelized even if the target tables have not been created or altered with PARALLEL specified. If you do not specify the PARALLEL hint, then the insert operation will not be parallelized unless all target tables were created or altered with PARALLEL specified.

See Also:

- "Hints" on page 2-92 and *Oracle9i Database Performance Tuning Guide and Reference* for the syntax and description of hints
- "Restrictions on multitable inserts" on page 17-65

single_table_insert

In a **single-table insert**, you insert values into one row of a table, view, or materialized view by specifying values explicitly or by retrieving the values through a subquery.

You can use the *flashback_clause* in *subquery* to insert past data into *table*.

See Also: the *flashback_clause* of SELECT on page 18-14 for more information on this clause

Restriction on single-table inserts: If you retrieve values through a subquery, then the select list of the subquery must have the same number of columns as the column list of the INSERT statement. If you omit the column list, then the subquery must provide values for every column in the table.

See Also: "Inserting Values into Tables: Examples" on page 17-66

insert_into_clause

Use the INSERT INTO clause to specify the target object or objects into which Oracle is to insert data.

DML_table_expression_clause

Use the INTO *dml_table_expression_clause* to specify the objects into which data is being inserted.

schema Specify the schema containing the table, view, or materialized view. If you omit *schema*, then Oracle assumes the object is in your own schema.

table* | *view* | *subquery Specify the name of the table or object table, view or object view, materialized view, or the column or columns returned by a subquery, into which rows are to be inserted. If you specify a view or object view, then Oracle inserts rows into the view's base table.

If any value to be inserted is a REF to an object table, and if the object table has a primary key object identifier, then the column into which you insert the REF must be a REF column with a referential integrity or SCOPE constraint to the object table.

If *table* (or the base table of *view*) contains one or more domain index columns, then this statement executes the appropriate indextype insert routine.

Issuing an INSERT statement against a table fires any INSERT triggers defined on the table.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for more information on these routines

PARTITION (*partition_name*) | SUBPARTITION (*subpartition_name*) Specify the name of the partition or subpartition within *table* (or the base table of *view*) targeted for inserts.

If a row to be inserted does not map into a specified partition or subpartition, then then Oracle returns an error.

Restriction on target partitions and subpartitions: This clause is not valid for object tables or object views.

dblink Specify a complete or partial name of a database link to a remote database where the table or view is located. You can insert rows into a remote table or view only if you are using Oracle's distributed functionality.

If you omit *dblink*, then Oracle assumes that the table or view is on the local database.

> **See Also:** "Syntax for Schema Objects and Parts in SQL Statements" on page 2-116 for information on referring to database links and "Inserting into a Remote Database: Example" on page 17-67

subquery_restriction_clause Use the *subquery_restriction_clause* to restrict the subquery in one of the following ways:

WITH READ ONLY Specify WITH READ ONLY to indicate that the table or view cannot be updated.

WITH CHECK OPTION Specify WITH CHECK OPTION to indicate that Oracle prohibits any changes to the table or view that would produce rows that are not included in the subquery.

CONSTRAINT *constraint* Specify the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

> **See Also:** "Using the WITH CHECK OPTION Clause: Example" on page 18-34

table_collection_expression The *table_collection_expression* lets you inform Oracle that the value of *collection_expression* should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value (that is, a value whose type is nested table or varray). This process of extracting the elements of a collection is called **collection unnesting**.

Note: In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as "THE subquery". That usage is now deprecated.

> **See Also:** "Table Collections: Examples" on page 18-38

t_alias

Specify a **correlation name** (alias) for the table, view, or subquery to be referenced elsewhere in the statement.

Restriction on table aliases: You cannot specify *t_alias* during a multitable insert.

Restrictions on the *dml_table_expression_clause*:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- With regard to the ORDER BY clause of the *subquery* in the *dml_table_expression_clause*, ordering is guaranteed only for the rows being inserted, and only within each extent of the table. Ordering of new rows with respect to existing rows is not guaranteed.

- If a view was created using the WITH CHECK OPTION, then you can insert into the view only rows that satisfy the view's defining query.
- If a view was created using a single base table, then you can insert rows into the view and then retrieve those values using the *returning_clause*.
- You cannot insert rows into a view except with INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list
  - Joins (with some exceptions as described in the paragraphs that follow).
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the INSERT statement will fail unless the SKIP_UNUSABLE_INDEXES session parameter has been set to TRUE.

  See Also: ALTER SESSION on page 10-2 for information on the SKIP_UNUSABLE_INDEXES session parameter

column

Specify a column of the table or view. In the inserted row, each column in this list is assigned a value from the *values_clause* or the subquery.

If you omit one or more of the table's columns from this list, then the column's value for the inserted row is the column's default value as specified when the table was created or last altered. If any omitted column has a NOT NULL constraint and no default value, then Oracle returns an error indicating that the constraint has been violated and rolls back the INSERT statement.

If you omit the column list altogether, then the *values_clause* or query must specify values for all columns in the table.

See Also: CREATE TABLE on page 15-7 for more information on default column values

values_clause

For a **single-table insert** operation, specify a row of values to be inserted into the table or view. You must specify a value in the *values_clause* for each column in the column list. If you omit the column list, then the *values_clause* must provide values for every column in the table.

For a **multitable insert** operation, each expression in the *values_clause* must refer to columns returned by the select list of the subquery. If you omit the *values_clause*, then the select list of the subquery determines the values to be inserted, so it must have the same number of columns as the column list of the corresponding *insert_into_clause*. If you do not specify a column list in the *insert_into_clause*, then the computed row must provide values for all columns in the target table.

For both types of insert operations, if you specify a column list in the *insert_into_clause*, then Oracle assigns to each column in the list a corresponding value from the values clause or the subquery. You can specify DEFAULT for any value in the *values_clause*. If you have specified a default value for the corresponding column of the table or view, then that value is inserted. If no default value for the corresponding column has been specified, then Oracle inserts null.

Note: Parallel direct-path INSERT supports only the subquery syntax of the INSERT statement, not the *values_clause*. Please refer to *Oracle9i Database Concepts* for information on serial and parallel direct-path INSERT.

Restrictions on inserted values:

- You cannot initialize an internal LOB attribute in an object with a value other than empty or null. That is, you cannot use a literal.
- You cannot insert a BFILE value until you have initialized the BFILE locator to null or to a directory alias and filename.
- When inserting into a list-partitioned table, you cannot insert a value into the partitioning key column that does not already exist in the *partition_value* list of one of the partitions.
- You cannot specify DEFAULT when inserting into a view.

Note: If you insert string literals into a RAW column, then during subsequent queries Oracle will perform a full table scan rather than using any index that might exist on the RAW column.

See Also:

- *Oracle Call Interface Programmer's Guide* and *Oracle9i Application Developer's Guide - Fundamentals* for information on initializing BFILE locators
- "About SQL Expressions" on page 4-2 and SELECT on page 18-4 for syntax of valid expressions
- "Using XML in SQL Statements" on page D-11 for information on inserting values into an XMLType table
- "Inserting into a BFILE: Example" on page 17-69, "Inserting into a Substitutable Tables and Columns: Examples", "Inserting Using the TO_LOB Function: Example" on page 17-69, "Inserting Sequence Values: Example" on page 17-68 and "Inserting Using Bind Variables: Example" on page 17-68

returning_clause

The returning clause retrieves the rows affected by a DML (INSERT, UPDATE, or DELETE) statement. You can specify this clause for tables and materialized views, and for views with a single base table.

When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and REFs to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the affected rows in bind arrays.

expr Each item in the *expr* list must be a valid expression syntax. All forms are valid except scalar subquery expressions.

INTO The INTO clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.

data_item Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the RETURNING list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the INTO list.

Restrictions:

- You cannot specify the *returning_clause* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.
- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

> **See Also:** *PL/SQL User's Guide and Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

multi_table_insert

In a **multitable insert**, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

> **Note:** Table aliases are not defined by the select list of the subquery. Therefore, they are not visible in the clauses dependent on the select list. For example, this can happen when trying to refer to an object column in an expression. To use an expression with a table alias, you must put the expression into the select list with a column alias, and then refer to the column alias in the VALUES clause or WHEN condition of the multitable insert

ALL *into_clause*

Specify ALL followed by multiple *insert_into_clauses* to perform an **unconditional multitable insert**. Oracle executes each *insert_into_clause* once for each row returned by the subquery.

conditional_insert_clause

Specify the *conditional_insert_clause* to perform a **conditional multitable insert**. Oracle filters each *insert_into_clause* through the corresponding WHEN condition, which determines whether that *insert_into_clause* is executed.

Each expression in the WHEN condition must refer to columns returned by the select list of the subquery. A single multitable insert statement can contain up to 127 WHEN clauses.

ALL If you specify ALL, then Oracle evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, Oracle executes the corresponding INTO clause list.

FIRST If you specify FIRST, then Oracle evaluates each WHEN clause in the order in which it appears in the statement. For the first WHEN clause that evaluates to true, Oracle executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

ELSE clause For a given row, if no WHEN clause evaluates to true:

- If you have specified an ELSE clause, then Oracle executes the INTO clause list associated with the ELSE clause.
- If you did not specify an else clause, then Oracle takes no action for that row.

See Also: "Multitable Inserts: Examples" on page 17-69

Restrictions on multitable inserts

- You can perform multitable inserts only on tables, not on views or materialized views.
- You cannot perform a multitable insert into a remote table.
- You cannot specify a table collection expression when performing a multitable insert.
- In a multitable insert, all of the *insert_into_clause*s cannot combine to specify more than 999 target columns.
- Multitable inserts are not parallelized in a Real Application Clusters environment, or if any target table is index organized, or if any target table has a bitmap index defined on it.
- Plan stability is not supported for multitable insert statements.
- The subquery of the multitable insert statement cannot use a sequence.

subquery

Specify a subquery that returns rows that are inserted into the table. The subquery can refer to any table, view, or materialized view, including the target tables of the INSERT statement. If the subquery selects no rows, then Oracle inserts no rows into the table.

You can use *subquery* in combination with the TO_LOB function to convert the values in a LONG column to LOB values in another column in the same or another table. To migrate LONGs to LOBs in a view, you must perform the migration on the base table, and then add the LOB to the view.

Notes on inserts using a subquery:

- If *subquery* returns (in part or totally) the equivalent of an existing materialized view, then Oracle may use the materialized view (for query rewrite) in place of one or more tables specified in *subquery*.

  See Also: *Oracle9i Data Warehousing Guide* for more information on materialized views and query rewrite

- If *subquery* refers to remote objects, then the INSERT operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *dml_table_expression_clause* refers to any remote objects, then the INSERT operation will run serially without notification. See *parallel_clause* on page 15-56 (in CREATE TABLE) for more information

  See Also:

  - "Inserting Values with a Subquery: Example" on page 17-67
  - "Inserting into a BFILE: Example" on page 17-69
  - *Oracle Call Interface Programmer's Guide* and *Oracle9i Application Developer's Guide - Fundamentals* for information on initializing BFILEs
  - "About SQL Expressions" on page 4-2 and SELECT on page 18-4 for syntax of valid expressions

Examples

Inserting Values into Tables: Examples The following statement inserts a row into the sample table departments:

```
INSERT INTO departments
   VALUES (280, 'Recreation', 121, 1700);
```

If the departments table had been created with a default value of 121 for the `manager_id` column, then you could issue the same statement as follows:

```
INSERT INTO departments
   VALUES (280, 'Recreation', DEFAULT, 1700);
```

The following statement inserts a row with six columns into the `employees` table. One of these columns is assigned `NULL` and another is assigned a number in scientific notation:

```
INSERT INTO employees (employee_id, last_name, email,
      hire_date, job_id, salary, commission_pct)
   VALUES (207, 'Gregory', 'pgregory@oracle.com',
      sysdate, 'PU_CLERK', 1.2E3, NULL);
```

The following statement has the same effect as the preceding example, but uses a subquery in the *`dml_table_expression_clause`*:

```
INSERT INTO
   (SELECT employee_id, last_name, email, hire_date, job_id,
      salary, commission_pct FROM employees)
   VALUES (207, 'Gregory', 'pgregory@oracle.com',
      sysdate, 'PU_CLERK', 1.2E3, NULL);
```

Inserting Values with a Subquery: Example The following statement copies employees whose commission exceeds 25% of their salary into the `bonuses` table (which is created in "Merging into a Table: Example" on page 17-80):

```
INSERT INTO bonuses
   SELECT employee_id, salary*1.1
   FROM employees
   WHERE commission_pct > 0.25 * salary;
```

Inserting into a Remote Database: Example The following statement inserts a row into the `accounts` table owned by the user `scott` on the database accessible by the database link `sales`:

```
INSERT INTO finance.accounts@sales (acc_no, acc_name)
   VALUES (5001, 'BOWER');
```

If that the `accounts` table has a `balance` column, then the newly inserted row is assigned the default value for this column (if one has been defined), because this `INSERT` statement does not specify a `balance` value.

Inserting Sequence Values: Example The following statement inserts a new row containing the next value of the departments sequence into the `departments` table:

```
INSERT INTO departments
   VALUES  (departments_seq.nextval, 'Entertainment', 162, 1400);
```

Inserting Using Bind Variables: Example The following example returns the values of the inserted rows into output bind variables :`bnd1` and :`bnd2`. (The bind variables must first be declared.)

```
INSERT INTO employees
      (employee_id, last_name, email, hire_date, job_id, salary)
   VALUES
   (employees_seq.nextval, 'Doe', 'john.doe@oracle.com',
       SYSDATE, 'SH_CLERK', 2400)
   RETURNING salary*12, job_id INTO :bnd1, :bnd2;
```

Inserting into a Substitutable Tables and Columns: Examples The following example inserts into the `persons` table, which is created in "Substitutable Table and Column Examples" on page 15-67. The first statement uses the root type `person_t`. The second insert uses the `employee_t` subtype `person_t`, and the third insert uses the `part_time_emp_t` subtype of `employee_t`:

```
INSERT INTO persons VALUES (person_t('Bob', 1234));
INSERT INTO persons VALUES (employee_t('Joe', 32456, 12, 100000));
INSERT INTO persons VALUES (
   part_time_emp_t('Tim', 5678, 13, 1000, 20));
```

The following example inserts into the `books` table, which was created in "Substitutable Table and Column Examples" on page 15-67. Notice that specification of the attribute values is identical to that for the substitutable table example:

```
INSERT INTO books VALUES (
   'An Autobiography', person_t('Bob', 1234));
INSERT INTO books VALUES (
   'Business Rules', employee_t('Joe', 3456, 12, 10000));
INSERT INTO books VALUES (
   'Mixing School and Work',
   part_time_emp_t('Tim', 5678, 13, 1000, 20));
```

You can extract data from substitutable tables and columns using built-in functions and conditions. For examples, see the functions TREAT on page 6-188 and SYS_TYPEID on page 6-161, and "IS OF type Conditions" on page 5-19.

Inserting Using the TO_LOB Function: Example The following example copies LONG data to a LOB column in the following long_tab table:

```
CREATE TABLE long_tab (pic_id NUMBER, long_pics LONG RAW);
```

First you must create a table with a LOB.

```
CREATE TABLE lob_tab (pic_id NUMBER, lob_pics BLOB);
```

Next, use an INSERT ... SELECT statement to copy the data in all rows for the LONG column into the newly created LOB column:

```
INSERT INTO lob_tab
   SELECT pic_id, TO_LOB(long_pics) FROM long_tab;
```

Once you are confident that the migration has been successful, you can drop the long_pics table. Alternatively, if the table contains other columns, then you can simply drop the LONG column from the table as follows:

```
ALTER TABLE long_tab DROP COLUMN long_pics;
```

Inserting into a BFILE: Example The following example inserts a row into the sample table pm.print_media. The example uses the BFILENAME function to identify a binary file on the server's file system:

```
CREATE DIRECTORY media_dir AS '/demo/schema/product_media';

INSERT INTO print_media (product_id, ad_id, ad_graphic)
   VALUES (3000, 31001,
      bfilename('media_dir', 'modem_comp_ad.gif'));
```

Multitable Inserts: Examples The following example uses the multitable insert syntax to insert into the sample table sh.sales some data from an input table with a different structure.

> **Note:** A number of constraints on the sales table have been disabled for purposes of this example, because the example ignores a number of table columns for the sake of brevity.

The input table looks like this:

```
SELECT * FROM sales_input_table;

PRODUCT_ID CUSTOMER_ID WEEKLY_ST  SALES_SUN  SALES_MON  SALES_TUE  SALES_WED SALES_THU  SALES_FRI  SALES_SAT
```

```
---------- ---------- --------- ---------- ---------- ---------- ------------------- ---------- ----------
       111        222 01-OCT-00        100        200        300        400        500        600        700
       222        333 08-OCT-00        200        300        400        500        600        700        800
       333        444 15-OCT-00        300        400        500        600        700        800        900
```

The multitable insert statement looks like this:

```
INSERT ALL
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date, sales_sun)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+1, sales_mon)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+2, sales_tue)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+3, sales_wed)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+4, sales_thu)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+5, sales_fri)
      INTO sales (prod_id, cust_id, time_id, amount)
      VALUES (product_id, customer_id, weekly_start_date+6, sales_sat)
   SELECT product_id, customer_id, weekly_start_date, sales_sun,
      sales_mon, sales_tue, sales_wed, sales_thu, sales_fri, sales_sat
      FROM sales_input_table;
```

Assuming these are the only rows in the `sales` table, the contents now look like this:

```
SELECT * FROM sales;

   PROD_ID    CUST_ID TIME_ID   C   PROMO_ID QUANTITY_SOLD     AMOUNT       COST
---------- ---------- --------- - ---------- ------------- ---------- ----------
       111        222 01-OCT-00                                  100
       111        222 02-OCT-00                                  200
       111        222 03-OCT-00                                  300
       111        222 04-OCT-00                                  400
       111        222 05-OCT-00                                  500
       111        222 06-OCT-00                                  600
       111        222 07-OCT-00                                  700
       222        333 08-OCT-00                                  200
       222        333 09-OCT-00                                  300
       222        333 10-OCT-00                                  400
       222        333 11-OCT-00                                  500
       222        333 12-OCT-00                                  600
       222        333 13-OCT-00                                  700
```

```
222        333 14-OCT-00                                  800
333        444 15-OCT-00                                  300
333        444 16-OCT-00                                  400
333        444 17-OCT-00                                  500
333        444 18-OCT-00                                  600
333        444 19-OCT-00                                  700
333        444 20-OCT-00                                  800
333        444 21-OCT-00                                  900
```

The next examples insert into multiple tables. Suppose you want to provide to sales representatives some information on orders of various sizes. The following example creates tables for small, medium, large, and "special" (very large) orders and populates those tables with data from the sample table `oe.orders`:

```
CREATE TABLE small_orders
   (order_id       NUMBER(12)   NOT NULL,
    customer_id    NUMBER(6)    NOT NULL,
    order_total    NUMBER(8,2),
    sales_rep_id   NUMBER(6)
   );

CREATE TABLE medium_orders AS SELECT * FROM small_orders;

CREATE TABLE large_orders AS SELECT * FROM small_orders;

CREATE TABLE special_orders
   (order_id       NUMBER(12)    NOT NULL,
    customer_id    NUMBER(6)     NOT NULL,
    order_total    NUMBER(8,2),
    sales_rep_id   NUMBER(6),
    credit_limit   NUMBER(9,2),
    cust_email     VARCHAR2(30)
   );
```

The first multitable insert populates only the tables for small, medium, and large orders:

```
INSERT ALL
   WHEN order_total < 1000000 THEN
      INTO small_orders
   WHEN order_total > 1000000 AND order_total < 2000000 THEN
      INTO medium_orders
   WHEN order_total > 2000000 THEN
      INTO large_orders
   SELECT order_id, order_total, sales_rep_id, customer_id
```

```
      FROM orders;
```

You can accomplish the same thing using the ELSE clause in place of the insert into the large_orders table:

```
INSERT ALL
   WHEN order_total < 100000 THEN
      INTO small_orders
   WHEN order_total > 100000 AND order_total < 200000 THEN
      INTO medium_orders
   ELSE
      INTO large_orders
   SELECT order_id, order_total, sales_rep_id, customer_id
      FROM orders;
```

The next example inserts into the small, medium, and large tables, as in the preceding example, and also puts orders greater than 2,900,000 into the special_orders table. This table also shows how to use column aliases to simplify the statement:

```
INSERT ALL
   WHEN ottl < 100000 THEN
      INTO small_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 100000 and ottl < 200000 THEN
      INTO medium_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 200000 THEN
      into large_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 290000 THEN
      INTO special_orders
   SELECT o.order_id oid, o.customer_id cid, o.order_total ottl,
      o.sales_rep_id sid, c.credit_limit cl, c.cust_email cem
      FROM orders o, customers c
      WHERE o.customer_id = c.customer_id;
```

Finally, the next example uses the FIRST clause to put orders greater than 2,900,000 into the special_orders table and *exclude* those orders from the large_orders table:

```
INSERT FIRST
   WHEN ottl < 100000 THEN
      INTO small_orders
         VALUES(oid, ottl, sid, cid)
   WHEN ottl > 100000 and ottl < 200000 THEN
```

```
   INTO medium_orders
      VALUES(oid, ottl, sid, cid)
WHEN ottl > 290000 THEN
   INTO special_orders
WHEN ottl > 200000 THEN
   INTO large_orders
      VALUES(oid, ottl, sid, cid)
SELECT o.order_id oid, o.customer_id cid, o.order_total ottl,
   o.sales_rep_id sid, c.credit_limit cl, c.cust_email cem
   FROM orders o, customers c
   WHERE o.customer_id = c.customer_id;
```

LOCK TABLE

Purpose

Use the LOCK TABLE statement to lock one or more tables (or table partitions or subpartitions) in a specified mode. This lock manually overrides automatic locking and permits or denies access to a table or view by other users for the duration of your operation.

Some forms of locks can be placed on the same table at the same time. Other locks allow only one lock for a table.

A locked table remains locked until you either commit your transaction or roll it back, either entirely or to a savepoint before you locked the table.

A lock never prevents other users from querying the table. A query never places a lock on a table. Readers never block writers and writers never block readers.

See Also:

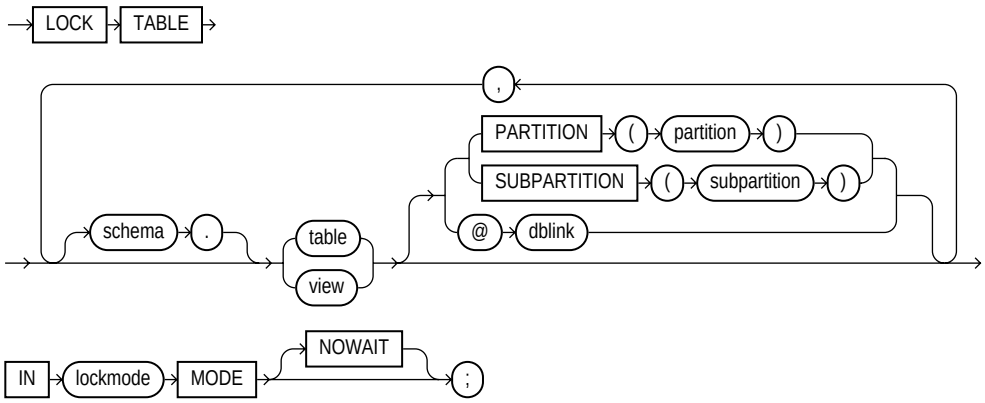
- *Oracle9i Database Concepts* for a complete description of the interaction of lock modes
- COMMIT on page 12-72
- ROLLBACK on page 17-100
- SAVEPOINT on page 18-2

Prerequisites

The table or view must be in your own schema or you must have the LOCK ANY TABLE system privilege, or you must have any object privilege on the table or view.

Syntax

lock_table::=



Keywords and Parameters

schema

Specify the schema containing the table or view. If you omit *schema*, then Oracle assumes the table or view is in your own schema.

table / *view*

Specify the name of the table to be locked.

If you specify *view*, then Oracle locks the view's base tables.

If you specify PARTITION (*partition*) or SUBPARTITION (*subpartition*), then Oracle first acquires an implicit lock on the table. The table lock is the same as the lock you specify for *partition* or *subpartition*, with two exceptions:

- If you specify a SHARE lock for the subpartition, then Oracle acquires an implicit ROW SHARE lock on the table.
- If you specify an EXCLUSIVE lock for the subpartition, then Oracle acquires an implicit ROW EXCLUSIVE lock on the table.

If you specify PARTITION and *table* is composite-partitioned, then Oracle acquires locks on all the subpartitions of *partition*.

Restriction on locking tables: If *table* is part of a hierarchy, then it must be the root of the hierarchy.

dblink

Specify a database link to a remote Oracle database where the table or view is located. You can lock tables and views on a remote database only if you are using Oracle's distributed functionality. All tables locked by a LOCK TABLE statement must be on the same database.

If you omit *dblink*, then Oracle assumes the table or view is on the local database.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-118 for information on specifying database links

lockmode Clause

Specify one of the following modes:

ROW SHARE ROW SHARE permits concurrent access to the locked table, but prohibits users from locking the entire table for exclusive access. ROW SHARE is synonymous with SHARE UPDATE, which is included for compatibility with earlier versions of Oracle.

ROW EXCLUSIVE ROW EXCLUSIVE is the same as ROW SHARE, but also prohibits locking in SHARE mode. ROW EXCLUSIVE locks are automatically obtained when updating, inserting, or deleting.

SHARE UPDATE See ROW SHARE.

SHARE SHARE permits concurrent queries but prohibits updates to the locked table.

SHARE ROW EXCLUSIVE SHARE ROW EXCLUSIVE is used to look at a whole table and to allow others to look at rows in the table but to prohibit others from locking the table in SHARE mode or updating rows.

EXCLUSIVE EXCLUSIVE permits queries on the locked table but prohibits any other activity on it.

NOWAIT

Specify NOWAIT if you want Oracle to return control to you immediately if the specified table (or specified partition or subpartition) is already locked by another user. In this case, Oracle returns a message indicating that the table, partition, or subpartition is already locked by another user.

If you omit this clause, then Oracle waits until the table is available, locks it, and returns control to you.

Examples

Locking a Table: Example The following statement locks the `employees` table in exclusive mode, but does not wait if another user already has locked the table:

```
LOCK TABLE employees
   IN EXCLUSIVE MODE
   NOWAIT;
```

The following statement locks the remote `accounts` table that is accessible through the database link `boston`:

```
LOCK TABLE accounts@boston
   IN SHARE MODE;
```

MERGE

Purpose

Use the MERGE statement to select rows from one table for update or insertion into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause.

This statement is a convenient way to combine at least two operations. It lets you avoid multiple INSERT and UPDATE DML statements.

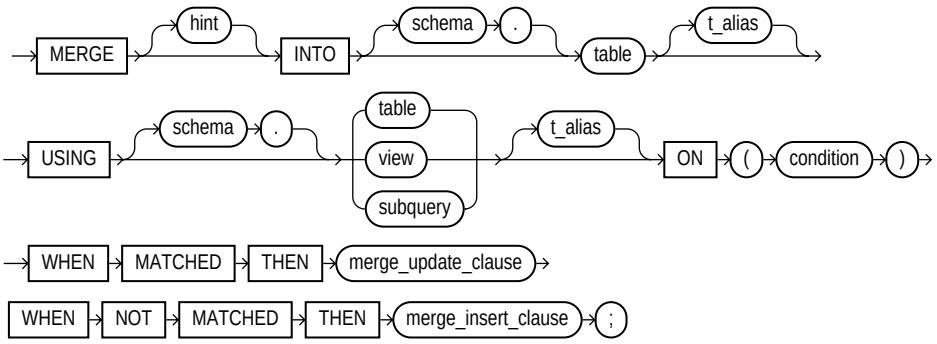
MERGE is a deterministic statement. That is, you cannot update the same row of the target table multiple times in the same MERGE statement.

Prerequisites

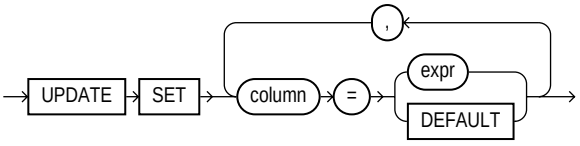
You must have INSERT and UPDATE object privileges on the target table and SELECT privilege on the source table.

Syntax

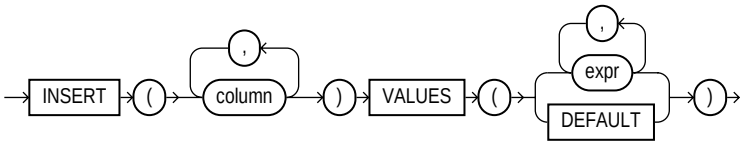
merge::=



merge_update_clause::=



merge_insert_clause::=



Keywords and Parameters

INTO Clause

Use the `INTO` clause to specify the target table you are updating or inserting into.

USING Clause

Use the `USING` clause to specify the source of the data to be updated or inserted. The source can be a table, view, or the result of a subquery.

ON Clause

Use the `ON` clause to specify the condition upon which the `MERGE` operation either updates or inserts. For each row in the target table for which the search condition is true, Oracle updates the row based with corresponding data from the source table. If the condition is not true for any rows, then Oracle inserts into the target table based on the corresponding source table row.

WHEN MATCHED | NOT MATCHED

Use these clauses to instruct Oracle how to respond to the results of the join condition in the `ON` clause. You can specify these two clauses in either order.

merge_update_clause

The *merge_update_clause* specifies the new column values of the target table. Oracle performs this update if the condition of the `ON` clause is true. If the update clause is executed, then all update triggers defined on the target table are activated.

Restriction on updating a view: You cannot specify `DEFAULT` when updating a view.

merge_insert_clause

The *merge_insert_clause* specifies values to insert into the column of the target table if the condition of the `ON` clause is false. If the insert clause is executed, then all insert triggers defined on the target table are activated.

Restriction on merging into a view: You cannot specify DEFAULT when updating a view.

Examples

Merging into a Table: Example The following example creates a bonuses table in the sample schema oe with a default bonus of 100. It then inserts into the bonuses table all employees who made sales (based on the sales_rep_id column of the oe.orders table). Finally, the Human Resources manager decides that *all* employees should receive a bonus. Those who have not made sales get a bonus of 1% of their salary. Those who already made sales get an increase in their bonus equal to 1% of their salary. The MERGE statement implements these changes in one step:

```
CREATE TABLE bonuses (employee_id NUMBER, bonus NUMBER DEFAULT 100);

INSERT INTO bonuses(employee_id)
   (SELECT e.employee_id FROM employees e, orders o
   WHERE e.employee_id = o.sales_rep_id
   GROUP BY e.employee_id);

SELECT * FROM bonuses;

EMPLOYEE_ID      BONUS
----------- ----------
        153        100
        154        100
        155        100
        156        100
        158        100
        159        100
        160        100
        161        100
        163        100

MERGE INTO bonuses D
   USING (SELECT employee_id, salary, department_id FROM employees
   WHERE department_id = 80) S
   ON (D.employee_id = S.employee_id)
   WHEN MATCHED THEN UPDATE SET D.bonus = D.bonus + S.salary*.01
   WHEN NOT MATCHED THEN INSERT (D.employee_id, D.bonus)
   VALUES (S.employee_id, S.salary*0.1);

EMPLOYEE_ID      BONUS
```

```
---------- ----------
       153        180
       154        175
       155        170
       156        200
       158        190
       159        180
       160        175
       161        170
       163        195
       157        950
       145       1400
       170        960
       179        620
       152        900
       169       1000
⋮
```

NOAUDIT

Purpose

Use the NOAUDIT statement to stop auditing previously enabled by the AUDIT statement.

The NOAUDIT statement must have the same syntax as the previous AUDIT statement. Further, it reverses the effects only of that particular statement. For example, suppose one AUDIT statement (statement A) enables auditing for a specific user. A second (statement B) enables auditing for all users. A NOAUDIT statement to disable auditing for all users (statement C) reverses statement B. However, statement C leaves statement A in effect and continues to audit the user that statement A specified.

See Also: AUDIT on page 12-52 for more information on auditing

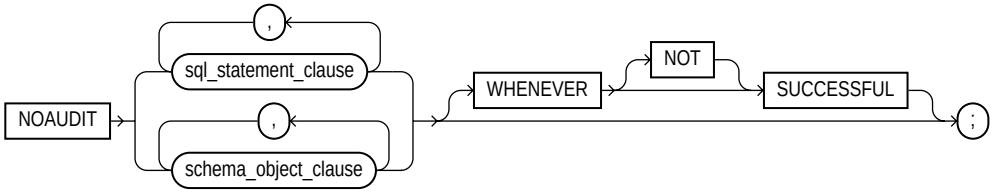
Prerequisites

To stop auditing of SQL statements, you must have the AUDIT SYSTEM system privilege.

To stop auditing of schema objects, you must be the owner of the object on which you stop auditing or you must have the AUDIT ANY system privilege. In addition, if the object you chose for auditing is a directory, then even if you created it, you must have the AUDIT ANY system privilege.

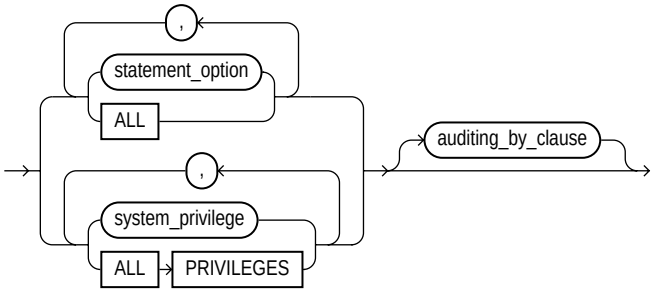
Syntax

noaudit::=

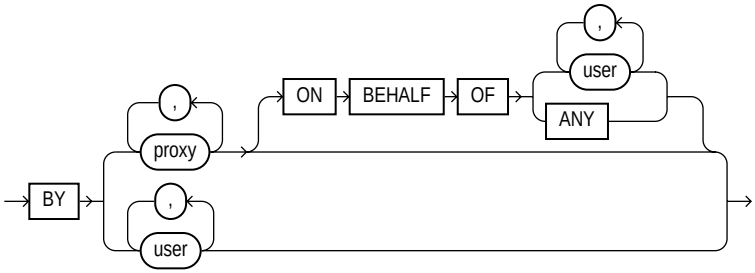


(*sql_statement_clause::=* on page 17-83, *schema_object_clause::=* on page 17-83)

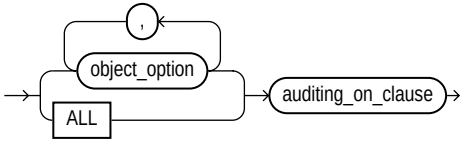
sql_statement_clause::=



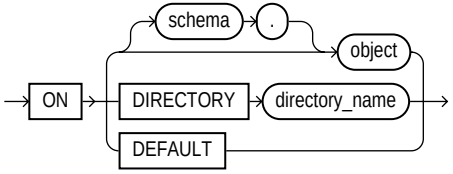
auditing_by_clause::=



schema_object_clause::=



auditing_on_clause::=



Keywords and Parameters

sql_statement_clause

Use the *sql_statement_clause* to stop auditing of a particular SQL statement.

statement_option For *statement_option*, specify the statement option for which auditing is to be stopped.

> **See Also:** Table 12–1 on page 12-58 and Table 12–2 on page 12-60 for a list of the statement options and the SQL statements they audit

ALL Specify ALL to stop auditing of all statement options currently being audited.

system_privilege For *system_privilege*, specify the system privilege for which auditing is to be stopped.

> **See Also:** Table 17–1 on page 17-36 for a list of the system privileges and the statements they authorize

ALL PRIVILEGES Specify ALL PRIVILEGES to stop auditing of all system privileges currently being audited.

auditing_by_clause Use the *auditing_by_clause* to stop auditing only those SQL statements issued by particular users. If you omit this clause, then Oracle stops auditing all users' statements.

- Specify BY *user* to stop auditing only for SQL statements issued by the specified users in their subsequent sessions. If you omit this clause, then Oracle stops auditing for all users' statements, except for the situation described for WHENEVER SUCCESSFUL.
- Specify BY *proxy* to stop auditing only for the SQL statements issued by the specified proxy, on behalf of a specific user or any user.

schema_object_clause

Use the *schema_object_clause* to stop auditing of a particular database object.

object_option For *object_option*, specify the type of operation for which auditing is to be stopped on the object specified in the ON clause.

See Also: Table 12–3 on page 12-62 for a list of these options

ALL Specify ALL as a shortcut equivalent to specifying all object options applicable for the type of object.

auditing_on_clause The *auditing_on_clause* lets you specify the particular schema object for which auditing is to be stopped.

- For object, specify the object name of a table, view, sequence, stored procedure, function, or package, materialized view, or library. If you do not qualify *object* with *schema*, then Oracle assumes the object is in your own schema.

  See Also: AUDIT on page 12-52 for information on auditing specific schema objects

- The DIRECTORY clause lets you specify the name of the directory on which auditing is to be stopped.
- Specify DEFAULT to remove the specified object options as default object options for subsequently created objects.

WHENEVER [NOT] SUCCESSFUL Specify WHENEVER SUCCESSFUL to stop auditing only for SQL statements and operations on schema objects that complete successfully.

Specify WHENEVER NOT SUCCESSFUL to stop auditing only for statements and operations that result in Oracle errors.

If you omit this clause, then Oracle stops auditing for all statements or operations, regardless of success or failure.

Examples

Stop Auditing of SQL Statements Related to Roles: Example If you have chosen auditing for every SQL statement that creates or drops a role, then you can stop auditing of such statements by issuing the following statement:

```
NOAUDIT ROLE;
```

Stop Auditing of Updates or Queries on Objects Owned by a Particular User: Example If you have chosen auditing for any statement that queries or updates any table issued by the users hr and oe, then you can stop auditing for queries by hr by issuing the following statement:

```
NOAUDIT SELECT TABLE BY hr;
```

The preceding statement stops auditing only queries by hr, so Oracle continues to audit queries and updates by oe as well as updates by hr.

Stop Auditing of Statements Authorized by a Particular Object Privilege: Example To stop auditing on all statements that are authorized by DELETE ANY TABLE system privilege, issue the following statement:

```
NOAUDIT DELETE ANY TABLE;
```

Stop Auditing of Queries on a Particular Object: Example If you have chosen auditing for every SQL statement that queries the employees table in the schema hr, then you can stop auditing for such queries by issuing the following statement:

```
NOAUDIT SELECT
   ON hr.employees;
```

Stop Auditing of Queries that Complete Successfully: Example You can stop auditing for queries that complete successfully by issuing the following statement:

```
NOAUDIT SELECT
   ON hr.employees
   WHENEVER SUCCESSFUL;
```

This statement stops auditing only for successful queries. Oracle continues to audit queries resulting in Oracle errors.

RENAME

Purpose

Caution: You cannot roll back a RENAME statement.

Use the RENAME statement to rename a table, view, sequence, or private synonym.

- Oracle automatically transfers integrity constraints, indexes, and grants on the old object to the new object.
- Oracle invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

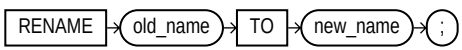
See Also: CREATE SYNONYM on page 15-2 and DROP SYNONYM on page 17-4

Prerequisites

The object must be in your own schema.

Syntax

rename::=



Keywords and Parameters

old_name

Specify the name of an existing table, view, sequence, or private synonym.

new_name

Specify the new name to be given to the existing object. The new name must not already be used by another schema object in the same namespace and must follow the rules for naming schema objects.

Restrictions on renaming objects:

- You cannot rename a public synonym. Instead, drop the public synonym and then re-create the public synonym with the new name.
- You cannot rename a type synonym that has any dependent tables or dependent valid user-defined object types.

See Also: "Schema Object Naming Rules" on page 2-111

Example

Renaming a Database Object: Example The following example uses a copy of the sample table `hr.departments`. To change the name of table `departments_new` to `emp_departments`, issue the following statement:

```
RENAME departments_new TO emp_departments;
```

You cannot use this statement directly to rename columns. However, you can rename a column using the ALTER TABLE ... *rename_column_clause*.

See Also: *rename_column_clause* on page 11-55

Another way to rename a column is to use the RENAME statement together with the CREATE TABLE statement with AS *subquery*. This method is useful is you are changing the structure of a table rather than only renaming a column. The following statements re-create the sample table `hr.job_history`, renaming a column from `department_id` to `dept_id`:

```
CREATE TABLE temporary
   (employee_id, start_date, end_date, job_id, dept_id)
AS SELECT
     employee_id, start_date, end_date, job_id, department_id
FROM job_history;

DROP TABLE job_history;

RENAME temporary TO job_history;
```

Note: Any integrity constraints defined on table *job_history* will be lost in the preceding example. You will have to redefine them on the new *job_history* table using an ALTER TABLE statement.

REVOKE

Purpose

Use the REVOKE statement to:

- Revoke system privileges from users and roles
- Revoke roles from users and roles
- Revoke object privileges for a particular object from users and roles

See Also:

- GRANT on page 17-29 for information on granting system privileges and roles
- Table 17–3 on page 17-47 for a summary of the object privileges for each type of object

Prerequisites

To revoke a **system privilege or role,** you must have been granted the privilege with the ADMIN OPTION.

To revoke a **role**, you must have been granted the role with the ADMIN OPTION. You can revoke any role if you have the GRANT ANY ROLE system privilege.

To revoke an **object privilege**, you must previously have granted the object privilege to the user and role or you must have the GRANT ANY OBJECT PRIVILEGE system privilege. In the latter case, you can revoke any object privilege that was granted by the object owner or on behalf of the owner (that is, by a user with the GRANT ANY OBJECT PRIVILEGE). However, you cannot revoke an object privilege that was granted by way of a WITH GRANT OPTION grant.

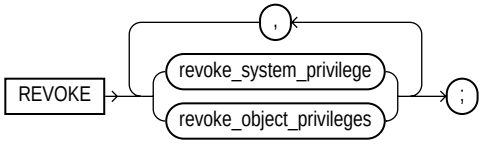
See Also: "Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example" on page 17-99

The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revokee
- Roles or object privileges granted through the operating system
- Privileges or roles granted to the revokee through roles

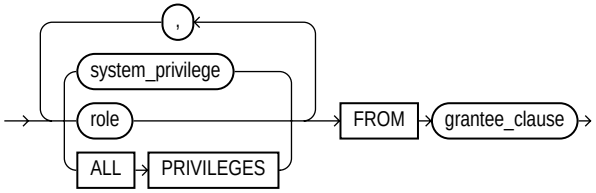
Syntax

revoke::=



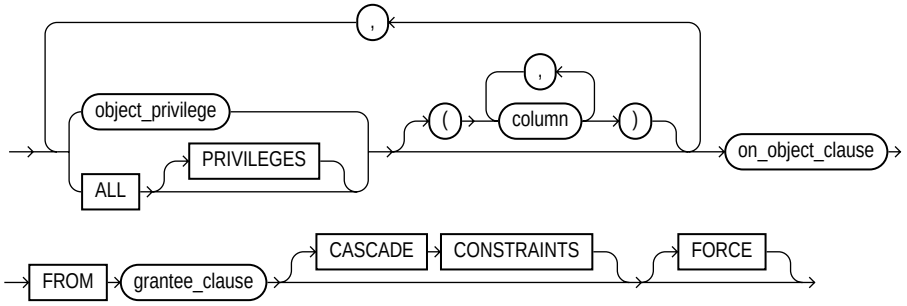
(*revoke_system_privileges::=* on page 17-90, *revoke_object_privileges::=* on page 17-90)

revoke_system_privileges::=



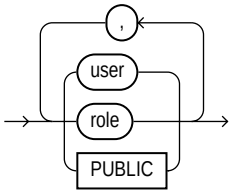
(*grantee_clause::=* on page 17-91)

revoke_object_privileges::=

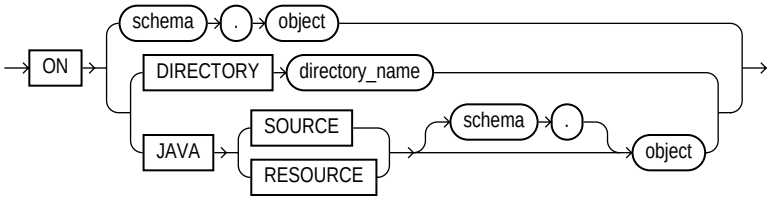


(*on_object_clause::=* on page 17-91, *grantee_clause::=* on page 17-91)

grantee_clause::=



on_object_clause::=



Keywords and Parameters

revoke_system_privileges

system_privilege

Specify the system privilege to be revoked.

> **See Also:** Table 17–1 on page 17-36 for a list of the system privileges

- If you revoke a privilege from a **user**, then Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

  > **See Also:** "Revoking a System Privilege from a User: Example" on page 17-96

- If you revoke a privilege from a **role**, then Oracle removes the privilege from the role's privilege domain. Effective immediately, users with the role enabled cannot exercise the privilege. Also, other users who have been granted the role and subsequently enable the role cannot exercise the privilege.

> **See Also:** "Revoking a System Privilege from a Role: Example" on page 17-96

- If you revoke a privilege from **PUBLIC**, then Oracle removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, such users can no longer exercise the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Restriction on system privileges: A system privilege cannot appear more than once in the list of privileges to be revoked.

Oracle provides a shortcut for specifying all system privileges at once:

- ALL PRIVILEGES: Specify ALL PRIVILEGES to revoke all the system privileges listed in Table 17–1 on page 17-36.

role

Specify the role to be revoked.

- If you revoke a role from a **user**, then Oracle makes the role unavailable to the user. If the role is currently enabled for the user, the user can continue to exercise the privileges in the role's privilege domain as long as it remains enabled. However, the user cannot subsequently enable the role.

  > **See Also:** "Revoking a Role from a User: Example" on page 17-96

- If you revoke a role from another **role**, then Oracle removes the revoked role's privilege domain from the revokee role's privilege domain. Users who have been granted and have enabled the revokee role can continue to exercise the privileges in the revoked role's privilege domain as long as the revokee role remains enabled. However, other users who have been granted the revokee role and subsequently enable it cannot exercise the privileges in the privilege domain of the revoked role.

  > **See Also:** "Revoking a Role from a Role: Example" on page 17-96

- If you revoke a role from **PUBLIC**, then Oracle makes the role unavailable to all users who have been granted the role through PUBLIC. Any user who has enabled the role can continue to exercise the privileges in its privilege domain as long as it remains enabled. However, users cannot subsequently enable the role. The role is not revoked from users who have been granted the role directly or through other roles.

Restriction on system roles: A system role cannot appear more than once in the list of roles to be revoked.

> **See Also:** Table 17–2 on page 17-46 for a list of the roles predefined by Oracle

grantee_clause

FROM *grantee_clause* identifies users or roles from which the system privilege, role, or object privilege is to be revoked.

PUBLIC Specify PUBLIC to revoke the privileges or roles from all users.

revoke_object_privileges

object_privilege

Specify the object privilege to be revoked. You can substitute any of the following values: ALTER, DELETE, EXECUTE, INDEX, INSERT, READ, REFERENCES, SELECT, UPDATE.

> **Note:** Each privilege authorizes some operation. By revoking a privilege, you prevent the revokee from performing that operation. However, multiple users may grant the same privilege to the same user, role, or PUBLIC. To remove the privilege from the grantee's privilege domain, all grantors must revoke the privilege. If even one grantor does not revoke the privilege, then the grantee can still exercise the privilege by virtue of that grant.

If you revoke a privilege from a **user**, then Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

- If that user has granted that privilege to other users or roles, then Oracle also revokes the privilege from those other users or roles.
- If that user's schema contains a procedure, function, or package that contains SQL statements that exercise the privilege, then the procedure, function, or package can no longer be executed.
- If that user's schema contains a view on that object, then Oracle invalidates the view.

- If you revoke the REFERENCES privilege from a user who has exercised the privilege to define referential integrity constraints, then you must specify the CASCADE CONSTRAINTS clause.

  See Also: "Revoking an Object Privilege from a User: Example" on page 17-97

If you revoke a privilege from a **role**, then Oracle removes the privilege from the role's privilege domain. Effective immediately, users with the role enabled cannot exercise the privilege. Other users who have been granted the role cannot exercise the privilege after enabling the role.

If you revoke a privilege from PUBLIC, then Oracle removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, all such users are restricted from exercising the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

See Also: "Revoking Object Privileges from PUBLIC: Example" on page 17-97

Restriction on object privileges: A privilege cannot appear more than once in the list of privileges to be revoked. A user, a role, or PUBLIC cannot appear more than once in the FROM clause.

ALL [PRIVILEGES]

Specify ALL to revoke all object privileges that you have granted to the revokee. (The keyword PRIVILEGES is provided for semantic clarity and is optional.)

Note: If no privileges have been granted on the object, then Oracle takes no action and does not return an error.

See Also: "Revoking All Object Privileges from a User: Example" on page 17-97

CASCADE CONSTRAINTS

This clause is relevant only if you revoke the REFERENCES privilege or ALL [PRIVILEGES]. It drops any referential integrity constraints that the revokee has defined using the REFERENCES privilege (which might have been granted either explicitly or implicitly through a grant of ALL [PRIVILEGES]).

> **See Also:** "Revoking an Object Privilege with CASCADE CONSTRAINTS: Example" on page 17-98

FORCE

Specify FORCE to revoke the EXECUTE object privilege on user-defined type objects with table or type dependencies. You must use FORCE to revoke the EXECUTE object privilege on user-defined type objects with table dependencies.

If you specify FORCE, then all privileges will be revoked, but all dependent objects are marked INVALID, data in dependent tables becomes inaccessible, and all dependent function-based indexes are marked UNUSABLE. (Regranting the necessary type privilege will revalidate the table.)

> **See Also:** *Oracle9i Database Concepts* for detailed information about type dependencies and user-defined object privileges

on_object_clause

The *on_object_clause* identifies the objects on which privileges are to be revoked.

object Specify the object on which the object privileges are to be revoked. This object can be:

- A table, view, sequence, procedure, stored function, or package, materialized view
- A synonym for a table, view, sequence, procedure, stored function, package, materialized view, or user-defined type
- A library, indextype, or user-defined operator

If you do not qualify object with *schema*, then Oracle assumes the object is in your own schema.

> **See Also:** "Revoking an Object Privilege on a Sequence from a User: Example" on page 17-97

If you revoke the SELECT object privilege (with or without the GRANT OPTION) on the containing table or materialized view of a materialized view, then Oracle invalidates the materialized view.

If you revoke the SELECT object privilege (with or without the GRANT OPTION) on any of the master tables of a materialized view, then Oracle invalidates both the materialized view and its containing table or materialized view.

DIRECTORY *directory_name* Specify the directory object on which privileges are to be revoked. You cannot qualify *directory_name* with *schema*. The object must be a directory.

> **See Also:** CREATE DIRECTORY on page 13-46 and "Revoking an Object Privilege on a Directory from a User: Example" on page 17-99

JAVA SOURCE | RESOURCE The JAVA clause lets you specify a Java source or resource schema object on which privileges are to be revoked.

Examples

Revoking a System Privilege from a User: Example The following statement revokes the DROP ANY TABLE system privilege from the users hr and oe:

```
REVOKE DROP ANY TABLE
    FROM hr, oe;
```

The users hr and oe can no longer drop tables in schemas other than their own.

Revoking a Role from a User: Example The following statement revokes the role dw_manager from the user sh:

```
REVOKE dw_manager
    FROM sh;
```

sh can no longer enable the dw_manager role.

Revoking a System Privilege from a Role: Example The following statement revokes the CREATE TABLESPACE system privilege from the dw_manager role:

```
REVOKE CREATE TABLESPACE
   FROM dw_manager;
```

Enabling the dw_manager role no longer allows users to create tablespaces.

Revoking a Role from a Role: Example To revoke the role dw_user from the role dw_manager, issue the following statement:

```
REVOKE dw_user
  FROM dw_manager;
```

dw_user privileges are no longer granted to dw_manager

Revoking an Object Privilege from a User: Example You can grant DELETE, INSERT, SELECT, and UPDATE privileges on the table orders to the user hr with the following statement:

```
GRANT ALL
    ON orders TO hr;
```

To revoke the DELETE privilege on orders from hr, issue the following statement:

```
REVOKE DELETE
    ON orders FROM hr;
```

Revoking All Object Privileges from a User: Example To revoke the remaining privileges on orders that you granted to hr, issue the following statement:

```
REVOKE ALL
    ON orders FROM hr;
```

Revoking Object Privileges from PUBLIC: Example You can grant SELECT and UPDATE privileges on the view emp_details_view to all users by granting the privileges to the role PUBLIC:

```
GRANT SELECT, UPDATE
    ON emp_details_view TO public;
```

The following statement revokes UPDATE privilege on emp_details_view from all users:

```
REVOKE UPDATE
    ON emp_details_view FROM public;
```

Users can no longer update the emp_details_view view, although users can still query it. However, if you have also granted the UPDATE privilege on emp_details_view to any users, either directly or through roles, then these users retain the privilege.

Revoking an Object Privilege on a Sequence from a User: Example You can grant the user oe the SELECT privilege on the departments_seq sequence in the schema hr with the following statement:

```
GRANT SELECT
```

```
    ON hr.departments_seq TO oe;
```

To revoke the SELECT privilege on departments_seq from oe, issue the following statement:

```
REVOKE SELECT
    ON hr.departments_seq FROM oe;
```

However, if the user hr has also granted SELECT privilege on departments to sh, then sh can still use departments by virtue of hr's grant.

Revoking an Object Privilege with CASCADE CONSTRAINTS: Example You can grant oe the privileges REFERENCES and UPDATE on the employees table in the schema hr with the following statement:

```
GRANT REFERENCES, UPDATE
    ON hr.employees TO oe;
```

oe can exercise the REFERENCES privilege to define a constraint in his own dependent table that refers to the employees table in the schema hr:

```
CREATE TABLE dependent
(dependno   NUMBER,
 dependname VARCHAR2(10),
 employee   NUMBER
    CONSTRAINT in_emp REFERENCES hr.employees(employee_id) );
```

You can revoke the REFERENCES privilege on hr.employees from oe by issuing the following statement that contains the CASCADE CONSTRAINTS clause:

```
REVOKE REFERENCES
    ON hr.employees
    FROM oe
    CASCADE CONSTRAINTS;
```

Revoking oe's REFERENCES privilege on hr.employees causes Oracle to drop the in_emp constraint, because oe required the privilege to define the constraint.

However, if oe has also been granted the REFERENCES privilege on hr.employees by a user other than you, then Oracle does not drop the constraint. oe still has the privilege necessary for the constraint by virtue of the other user's grant.

Revoking an Object Privilege on a Directory from a User: Example You can revoke READ privilege on directory bfile_dir from hr, by issuing the following statement:

```
REVOKE READ ON DIRECTORY bfile_dir FROM hr;
```

Revoke Operations that Use GRANT ANY OBJECT PRIVILEGE: Example Suppose that the database administrator has granted GRANT ANY OBJECT PRIVILEGE to user sh. Now suppose that user hr grants the update privilege on the employees table to oe:

```
CONNECT hr/hr
GRANT UPDATE ON employees TO oe WITH GRANT OPTION;
```

This grant gives user oe the right to pass the object privilege along to another user:

```
CONNECT oe/oe
GRANT UPDATE ON hr.employees TO pm;
```

User sh, who has the GRANT ANY OBJECT PRIVILEGE, can now act on behalf of user hr and revoke the update privilege from user oe, because oe was granted the privilege by hr:

```
CONNECT sh/sh
REVOKE UPDATE ON hr.employees FROM oe;
```

User sh cannot revoke the update privilege from user pm explicitly, because pm received the grant neither from the object owner (hr), nor from sh, nor from another user with GRANT ANY OBJECT PRIVILEGE, but from user oe. However, the preceding statement cascades, removing all privileges that depend on the one revoked. Therefore the object privilege is implicitly revoked from pm as well.

ROLLBACK

Purpose

Use the ROLLBACK statement to undo work done in the current transaction, or to manually undo the work done by an in-doubt distributed transaction.

Note: Oracle recommends that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, then Oracle rolls back the last uncommitted transaction.

See Also:

- *Oracle9i Database Concepts* for information on transactions
- *Oracle9i Heterogeneous Connectivity Administrator's Guide* for information on distributed transactions
- SET TRANSACTION on page 18-50 for information on setting characteristics of the current transaction
- COMMIT on page 12-72
- SAVEPOINT on page 18-2

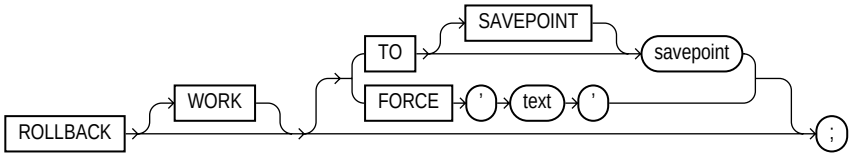
Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have the FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have the FORCE ANY TRANSACTION system privilege.

Syntax

rollback::=



Keywords and Parameters

WORK

The keyword `WORK` is optional and is provided for ANSI compatibility.

TO SAVEPOINT Clause

Specify the savepoint to which you want to roll back the current transaction. If you omit this clause, then the `ROLLBACK` statement rolls back the entire transaction.

Using `ROLLBACK` without the `TO SAVEPOINT` clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases the transaction's locks

See Also: SAVEPOINT on page 18-2

Using `ROLLBACK` with the `TO SAVEPOINT` clause performs the following operations:

- Rolls back just the portion of the transaction after the savepoint.
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- Releases all table and row locks acquired since the savepoint. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that

have not already requested the rows can request and access the rows immediately.

Restrictions on in-doubt transactions: You cannot manually roll back an in-doubt transaction to a savepoint.

FORCE Clause

Specify FORCE to manually roll back an in-doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.

A ROLLBACK statement with a FORCE clause rolls back only the specified transaction. Such a statement does not affect your current transaction.

Restriction on forcing rollback: ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

> **See Also:** *Oracle9i Heterogeneous Connectivity Administrator's Guide* for more information on distributed transactions and rolling back in-doubt transactions

Examples

Rolling Back Transactions: Examples The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

The following statement rolls back your current transaction to savepoint banda_sal:

```
ROLLBACK TO SAVEPOINT banda_sal;
```

> **See Also:** "Creating Savepoints: Example" on page 18-2 for a full version of this example

The following statement manually rolls back an in-doubt distributed transaction:

```
ROLLBACK WORK
    FORCE '25.32.87';
```

18

SQL Statements: SAVEPOINT to UPDATE

This chapter contains the following SQL statements:

- SAVEPOINT
- SELECT
- SET CONSTRAINT[S]
- SET ROLE
- SET TRANSACTION
- TRUNCATE
- UPDATE

SAVEPOINT

Purpose

Use the SAVEPOINT statement to identify a point in a transaction to which you can later roll back.

See Also:

- *Oracle9i Database Concepts* for information on savepoints.
- ROLLBACK on page 17-100 for information on rolling back transactions
- SET TRANSACTION on page 18-50 for information on setting characteristics of the current transaction

Prerequisites

None.

Syntax

savepoint::=

Keywords and Parameters

savepoint

Specify the name of the savepoint to be created.

Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, then the earlier savepoint is erased. After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

Example

Creating Savepoints: Example To update Banda's and Greene's salary in the sample table hr.employees, check that the total department salary does not exceed 314,000, then reenter Greene's salary, enter:

```
UPDATE employees
    SET salary = 7000
    WHERE last_name = 'Banda';
SAVEPOINT banda_sal;

UPDATE employees
    SET salary = 12000
    WHERE last_name = 'Greene';
SAVEPOINT greene_sal;

SELECT SUM(salary) FROM employees;

ROLLBACK TO SAVEPOINT banda_sal;

UPDATE employees
    SET salary = 11000
    WHERE last_name = 'Greene';

COMMIT;
```

SELECT

Purpose

Use a SELECT statement or subquery to retrieve data from one or more tables, object tables, views, object views, or materialized views.

Note: If the result (or part of the result) of a SELECT statement is equivalent to an existing materialized view, then Oracle may use the materialized view in place of one or more tables specified in the SELECT statement. This substitution is called **query rewrite**, and takes place only if cost optimization is enabled and the QUERY_REWRITE_ENABLED parameter is set to TRUE. To determine whether query write has occurred, use the EXPLAIN PLAN statement.

See Also:

- Chapter 8, "SQL Queries and Subqueries" for general information on queries and subqueries
- *Oracle9i Data Warehousing Guide* for more information on materialized views and query rewrite
- EXPLAIN PLAN on page 17-24

Prerequisites

For you to select data from a table or materialized view, the table or materialized view must be in your own schema or you must have the SELECT privilege on the table or materialized view.

For you to select rows from the base tables of a view,

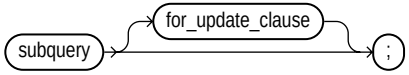
- You must have the SELECT privilege on the view, and
- Whoever owns the schema containing the view must have the SELECT privilege on the base tables.

The SELECT ANY TABLE system privilege also allows you to select data from any table or any materialized view or any view's base table.

To issue a flashback query (using the *flashback_clause*), either you must have FLASHBACK object privilege on the objects in the select list, or you must have FLASHBACK ANY TABLE system privilege.

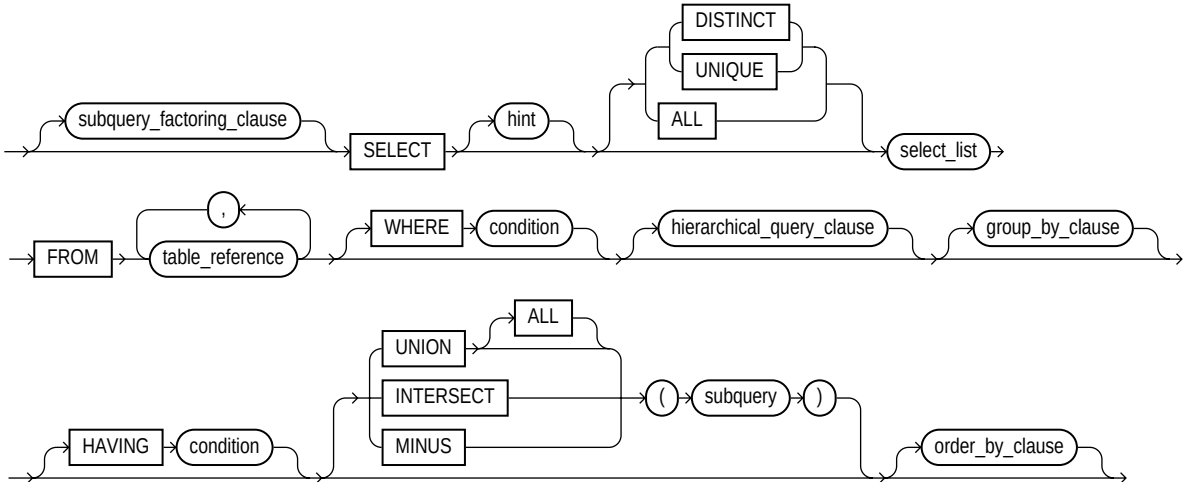
Syntax

select::=



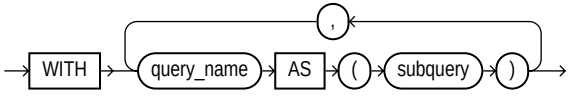
(*for_update_clause::=* on page 18-10)

subquery::=

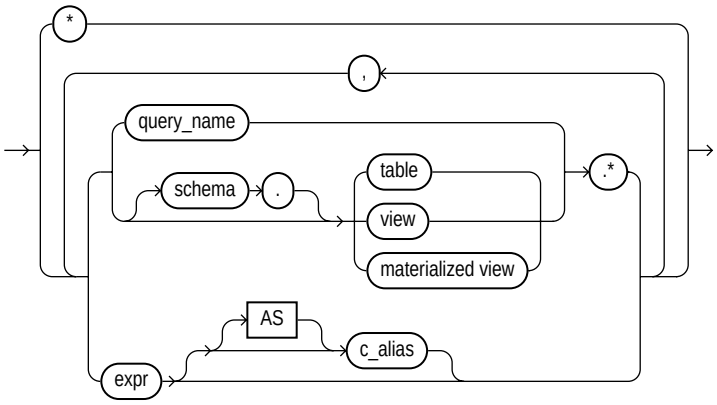


(*subquery_factoring_clause::=* on page 18-5, *select_list::=* on page 18-6, *table_reference::=* on page 18-6, *hierarchical_query_clause::=* on page 18-8, *group_by_clause::=* on page 18-8, *order_by_clause::=* on page 18-9)

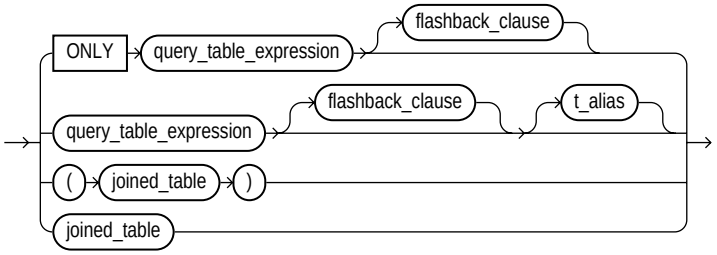
subquery_factoring_clause::=



select_list::=

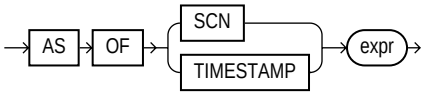


table_reference::=

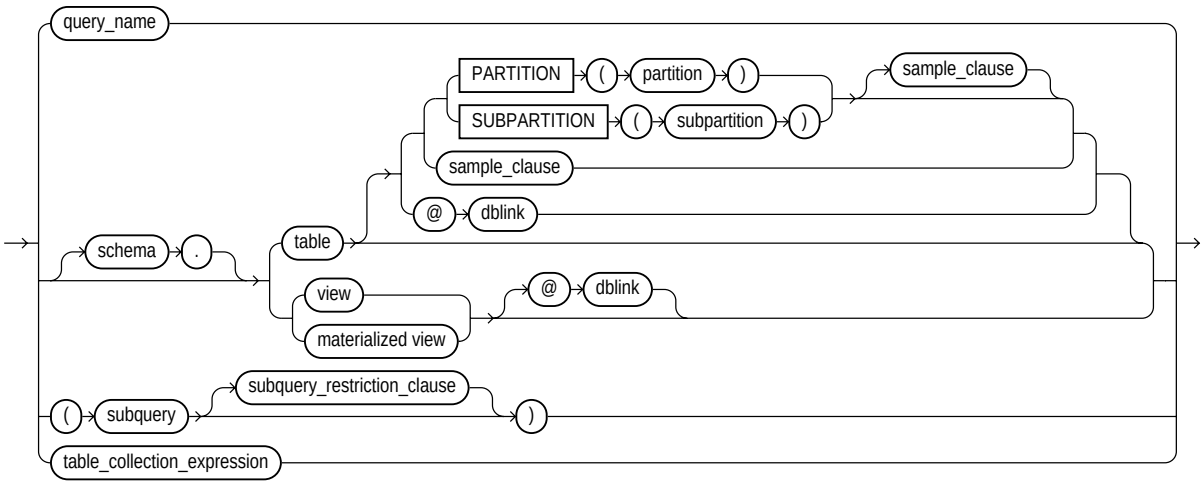


(*query_table_expression::=* on page 18-7, *flashback_clause::=* on page 18-6)

flashback_clause::=

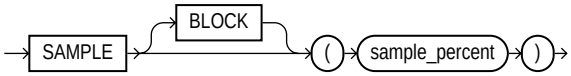


query_table_expression::=

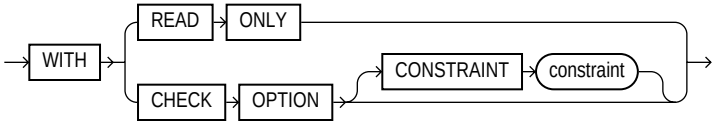


(*subquery_restriction_clause::=* on page 18-7, *table_collection_expression::=* on page 18-7)

sample_clause::=

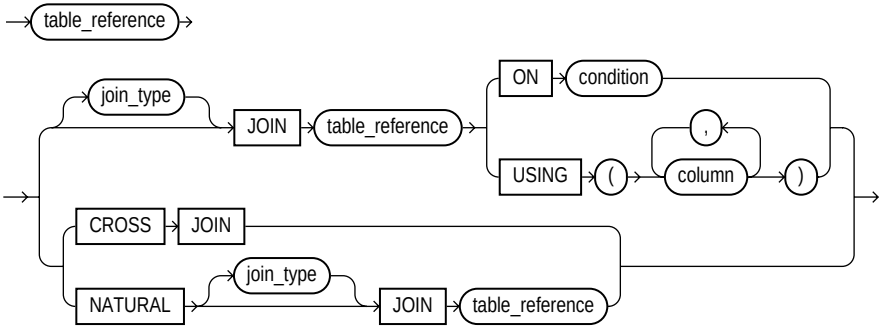


subquery_restriction_clause::=



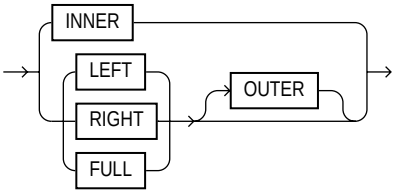
table_collection_expression::=

joined_table::=

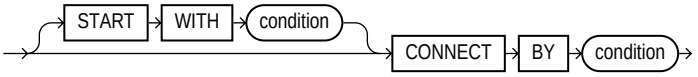


(*table_reference::=* on page 18-6)

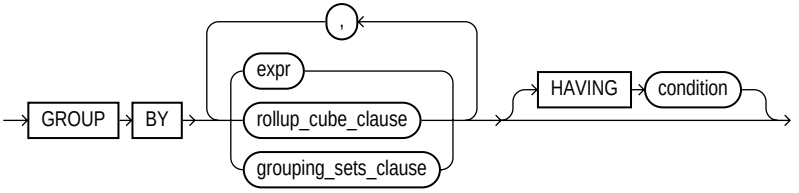
join_type::=



hierarchical_query_clause::=

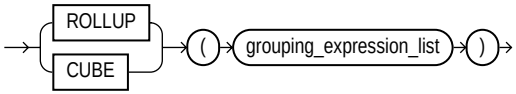


group_by_clause::=



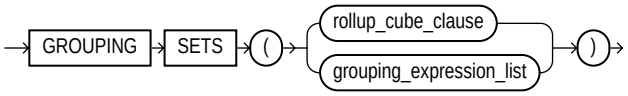
(*rollup_cube_clause::=* on page 18-9, *grouping_sets_clause::=* on page 18-9)

rollup_cube_clause::=



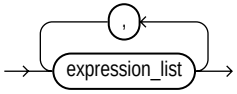
(*grouping_expression_list::=* on page 18-9)

grouping_sets_clause::=

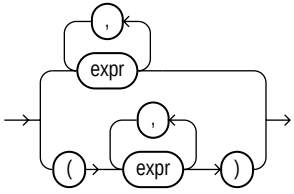


(*rollup_cube_clause::=* on page 18-9, *grouping_expression_list::=* on page 18-9)

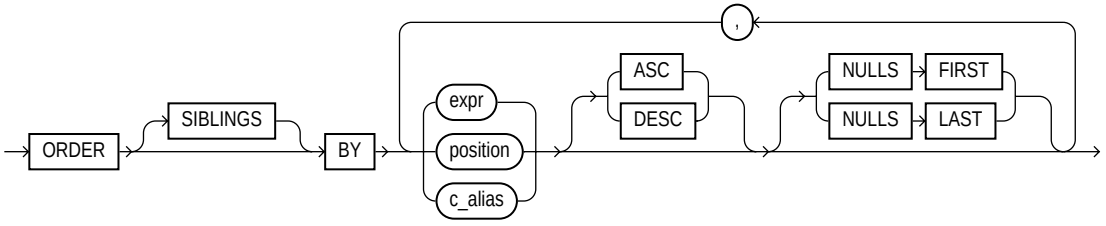
grouping_expression_list::=



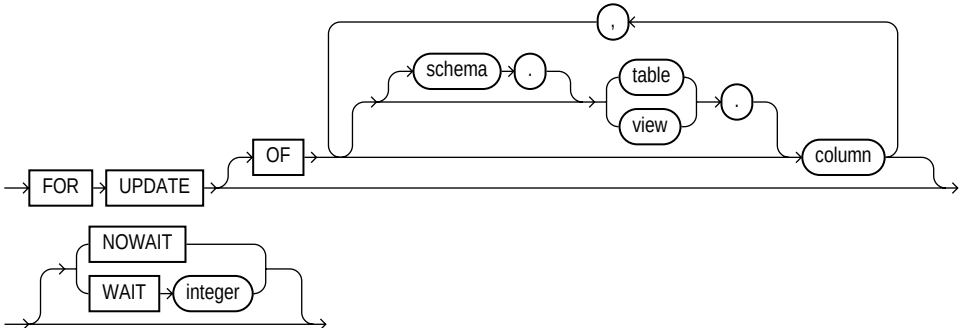
expression_list::=



order_by_clause::=



for_update_clause::=



Keywords and Parameters

subquery_factoring_clause

The *subquery_factoring_clause* (WITH *query_name*) lets you assign names to subquery blocks. You can then reference the subquery block multiple places in the query by specifying the query name. Oracle optimizes the query by treating the query name as either an inline view or as a temporary table.

You can specify this clause in any top-level SELECT statement and in most types of subqueries. The query name is visible to all subsequent subqueries (except the subquery that defines the query name itself) and to the main query.

Restrictions on subquery factoring:

- You cannot nest this clause. That is, you cannot specify the *subquery_factoring_clause* as a subquery within another *subquery_factoring_clause*.
- In a query with set operators, the set operator subquery cannot contain the *subquery_factoring_clause*, but the FROM subquery can contain the *subquery_factoring_clause*.

See Also:

- *Oracle9i Database Concepts* for information about inline views
- *Oracle9i Data Warehousing Guide* and *Oracle9i Application Developer's Guide - Fundamentals* for information on using the subquery factoring feature
- "Subquery Factoring: Example" on page 18-27

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

> **See Also:** "Hints" on page 2-92 and *Oracle9i Database Performance Tuning Guide and Reference* for the syntax and description of hints

DISTINCT | UNIQUE

Specify `DISTINCT` or `UNIQUE` if you want Oracle to return only one copy of each set of duplicate rows selected (these two keywords are synonymous). Duplicate rows are those with matching values for each expression in the select list.

Restrictions on DISTINCT and UNIQUE queries:

- When you specify `DISTINCT` or `UNIQUE`, the total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter `DB_BLOCK_SIZE`.
- You cannot specify `DISTINCT` if the *`select_list`* contains LOB columns.

ALL

Specify `ALL` if you want Oracle to return all rows selected, including all copies of duplicates. The default is `ALL`.

* [asterisk]

Specify the asterisk to select all columns from all tables, views, or materialized views listed in the `FROM` clause.

Note: If you are selecting from a table (that is, you specify a table in the `FROM` clause rather than a view or a materialized view), then then columns that have been marked as `UNUSED` by the `ALTER TABLE SET UNUSED` statement are not selected.

> **See Also:** ALTER TABLE on page 11-2, "Simple Query Examples" on page 18-28, and "Selecting from the DUAL Table: Example" on page 18-43

select_list

The *select_list* lets you specify the columns you want to retrieve from the database.

query_name

For *query_name*, specify a name already specified in the *subquery_factoring_clause*. You must have specified the *subquery_factoring_clause* in order to specify *query_name* in the *select_list*. If you specify *query_name* in the *select_list*, then you also must specify *query_name* in the *query_table_expression* (FROM clause).

table. | view.* | materialized view.**

Specify the object name followed by a period and the asterisk to select all columns from the specified table, view, or materialized view. A query that selects rows from two or more tables, views, or materialized views is a join.

You can use the schema qualifier to select from a table, view, or materialized view in a schema other than your own. If you omit *schema*, then Oracle assumes the table, view, or materialized view is in your own schema.

> **See Also:** "Joins" on page 8-9

expr

Specify an expression representing the information you want to select. A column name in this list can be qualified with *schema* only if the table, view, or materialized view containing the column is qualified with *schema* in the FROM clause. If you specify a member method of an object type, then you must follow the method name with parentheses even if the method takes no arguments.

> **See Also:** "Selecting Sequence Values: Examples" on page 18-43

c_alias Specify a different name (alias) for the column expression. Oracle will use this alias in the column heading. The AS keyword is optional. The alias effectively renames the select list item for the duration of the query. The alias can be used in the *order_by_clause*, but not other clauses in the query.

See Also:

- *Oracle9i Data Warehousing Guide* for information on using the *expr* AS *c_alias* syntax with the UNION ALL operator in queries of multiple materialized views
- "About SQL Expressions" on page 4-2 for the syntax of *expr*

Restrictions on the *select_list*:

- If you also specify a *group_by_clause* in this statement, then this select list can contain only the following types of expressions:
  - Constants
  - Aggregate functions and the functions USER, UID, and SYSDATE
  - Expressions identical to those in the *group_by_clause*
  - Expressions involving the preceding expressions that evaluate to the same value for all rows in a group
- You can select a rowid from a join view only if the join has one and only one key-preserved table. The rowid of that table becomes the rowid of the view.

  See Also: *Oracle9i Database Administrator's Guide* for information on key-preserved tables

- If two or more tables have some column names in common, then you must qualify column names with names of tables.

FROM Clause

The FROM clause lets you specify the objects from which data is selected.

query_table_expression

Use the *query_table_expression* clause to identify a table, view, materialized view, or partition, or to specify a subquery that identifies the objects.

See Also: "Using Subqueries: Examples" on page 18-35

ONLY The ONLY clause applies only to views. Specify ONLY if the view in the FROM clause is a view belonging to a hierarchy and you do not want to include rows from any of its subviews.

flashback_clause

Use the *flashback_clause* to query past data from a table, view, or materialized view. If you specify SCN, then *expr* must evaluate to a number. If you specify TIMESTAMP, then *expr* must evaluate to a timestamp value. Oracle returns rows as they existed at the specified system change number or time.

Note: This clause implements SQL-driven flashback, which lets you specify a different system change number or timestamp for each object in the select list. You can also implement session-level flashback using the DBMS_FLASHBACK package. For information on session-level flashback, please refer to *Oracle9i Application Developer's Guide - Fundamentals* and *Oracle9i Supplied PL/SQL Packages and Types Reference*.

Restrictions on the *flashback_clause*:

- You cannot apply the *flashback_clause* to a remote database object. However, you can include remote objects in a join with local objects to which you apply the *flashback_clause*.
- You cannot specify this clause if you have specify *query_name* in the *query_table_expression*.

See Also:

- *Oracle9i Application Developer's Guide - Fundamentals* for more information on flashback queries
- "Using Flashback Queries: Example" on page 18-29

PARTITION | SUBPARTITION For PARTITION or SUBPARTITION, specify the partition or subpartition from which you want to retrieve data. The *partition* parameter may be the name of the partition within *table* from which to retrieve data or a more complicated predicate restricting retrieval to just one partition of the table.

See Also: "Selecting from a Partition: Example" on page 18-28

dblink For *dblink*, specify the complete or partial name for a database link to a remote database where the table, view, or materialized view is located. This database need not be an Oracle database.

See Also:

- "Referring to Objects in Remote Databases" on page 2-118 for more information on referring to database links
- "Distributed Queries" on page 8-15 for more information about distributed queries and "Using Distributed Queries: Example" on page 18-42

If you omit *dblink*, then Oracle assumes that the table, view, or materialized view is on the local database.

Restrictions on database links:

- You cannot query a user-defined type or an object REF on a remote table.
- You cannot query columns of type AnyType, AnyData, or AnyDataSet from remote tables.

table* | *view* | *materialized view For *table*, *view*, or *materialized view*, specify the name of a table, view, or materialized view from which data is selected.

sample_clause

The *sample_clause* lets you instruct Oracle to select from a random sample of rows from the table, rather than from the entire table.

See Also: "Selecting a Sample: Examples" on page 18-29

BLOCK BLOCK instructs Oracle to perform random block sampling instead of random row sampling.

See Also: *Oracle9i Database Concepts* for a discussion of the difference

sample_percent *sample_percent* is a number specifying the percentage of the total row or block count to be included in the sample. The value must be in the range .000001 to (but not including) 100.

Restrictions on the *sample_clause*:

- You can specify SAMPLE only in a query that selects from a single table. Joins are not supported. However, you can achieve the same results by using a CREATE TABLE ... AS SELECT query to materialize a sample of an underlying table and then rewrite the original query to refer to the newly created table

sample. If you wish, you can write additional queries to materialize samples for other tables.

> **See Also:** "Selecting a Sample: Examples" on page 18-29

- When you specify SAMPLE, Oracle automatically uses cost-based optimization. Rule-based optimization is not supported with this clause.

> **Caution:** The use of statistically incorrect assumptions when using this feature can lead to incorrect or undesirable results.

subquery_restriction_clause The *subquery_restriction_clause* lets you restrict the subquery in one of the following ways:

WITH READ ONLY Specify WITH READ ONLY to indicate that the table or view cannot be updated.

WITH CHECK OPTION Specify WITH CHECK OPTION to indicate that Oracle prohibits any changes to the table or view that would produce rows that are not included in the subquery.

CONSTRAINT *constraint* Specify the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

> **See Also:** "Using the WITH CHECK OPTION Clause: Example" on page 18-34

table_collection_expression

The *table_collection_expression* lets you inform Oracle that the value of *collection_expression* should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value (that is, a value whose type is nested table or varray). This process of extracting the elements of a collection is called **collection unnesting**.

Note: In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as "THE subquery". That usage is now deprecated.

The *collection_expression* can reference columns of tables defined to its left in the FROM clause. This is called **left correlation**. Left correlation can occur only in *table_collection_expression*. Other subqueries cannot contains references to columns defined outside the subquery.

The optional "(+)" lets you specify that *table_collection_expression* should return a row with all fields set to NULL if the collection is null or empty. The "(+)" is valid only if *collection_expression* uses left correlation. The result is similar to that of an outer join.

Note: When you use the "(+)" syntax in the WHERE clause of a subquery in an UPDATE or DELETE operation, you must specify two tables in the FROM clause of the subquery. Oracle ignores the outer join syntax unless there is a join in the subquery itself.

See Also:

- "Outer Joins" on page 8-11
- "Table Collections: Examples" on page 18-38 and "Collection Unnesting: Examples" on page 18-39

t_alias

Specify a **correlation name** (alias) for the table, view, materialized view, or subquery for evaluating the query. Correlation names are most often used in a correlated query. Other references to the table, view, or materialized view throughout the query must refer to this alias.

Note: This alias is *required* if the *query_table_expr_clause* references any object type attributes or object type methods.

See Also: "Using Correlated Subqueries: Examples" on page 18-42

joined_table

Use the *joined_table* syntax to identify tables that are part of a join from which to select data.

> **See Also:** "Joins" on page 8-9 for more information on joins, "Using Join Queries: Examples" on page 18-34, "Using Self Joins: Example" on page 18-36, and "Using Outer Joins: Examples" on page 18-36

join_type The *join_type* indicates the kind of join being performed:

- Specify INNER to indicate explicitly that an inner join is being performed. This is the default.
- Specify RIGHT to indicate a right outer join.
- Specify LEFT to indicate a left outer join.
- Specify FULL to indicate a full or two-sided outer join. In addition to the inner join, rows from both tables that have not been returned in the result of the inner join will be preserved and extended with nulls.
- You can specify the optional OUTER keyword following RIGHT, LEFT, or FULL to explicitly clarify that an outer join is being performed.

JOIN The JOIN keyword explicitly states that a join is being performed. You can use this syntax to replace the comma-delimited table expressions used in WHERE clause joins with FROM clause join syntax.

ON *condition* Use the ON clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the WHERE clause.

USING *column* When you are specifying an equijoin of columns that have the same name in both tables, the USING column clause indicates the columns to be used. You can use this clause only if the join columns in both tables have the same name. Do not qualify the column name with a table name or table alias.

In an outer join with the USING clause, the query returns a single column which is a coalesce of the two matching columns in the join. The coalesce functions as follows:

```
COALESCE (a, b) = a if a NOT NULL, else b.
```

Therefore:

- A left outer join returns all the common column values from the left table in the FROM clause.
- A right outer join returns all the common column values from the right table in the FROM clause.
- A full outer join returns all the common column values from both joined tables.

Restriction on USING *column*: You cannot specify a LOB column or a collection column in the USING *column* clause.

See Also: "Using Outer Joins: Examples" on page 18-36

CROSS JOIN The CROSS keyword indicates that a cross join is being performed. A cross join produces the cross-product of two relations and is essentially the same as the comma-delimited Oracle notation.

NATURAL JOIN The NATURAL keyword indicates that a natural join is being performed. A natural join is based on all columns in the two tables that have the same name. It selects rows from the two tables that have equal values in the relevant columns. When specifying columns that are involved in the natural join, do not qualify the column name with a table name or table alias.

Note: On occasion, the table pairings in natural or cross joins may be ambiguous. For example:

```
a NATURAL LEFT JOIN b LEFT JOIN c ON b.c1 = c.c1
```

can be interpreted in either of the following ways:

```
 a NATURAL LEFT JOIN (b LEFT JOIN c ON b.c1 = c.c1)
(a NATURAL LEFT JOIN b) LEFT JOIN c ON b.c1 = c.c1
```

To avoid this ambiguity, you can use parentheses to specify the pairings of joined tables. In the absence of such parentheses, Oracle uses left associativity, pairing the tables from left to right.

Restriction on natural joins: You cannot specify a LOB column or a collection column as part of a natural join.

WHERE *condition*

The WHERE condition lets you restrict the rows selected to those that satisfy one or more conditions. For *condition*, specify any valid SQL condition.

> **See Also:** Chapter 5, "Conditions" for the syntax description of *condition*

If you omit this clause, then Oracle returns all rows from the tables, views, or materialized views in the FROM clause.

Note: If this clause refers to a DATE column of a partitioned table or index, then Oracle performs partition pruning only if (1) you created the table or index partitions by fully specifying the year using the TO_DATE function with a 4-digit format mask, *and* (2) you specify the date in the query's *where_clause* using the TO_DATE function and either a 2- or 4-digit format mask.

See Also: "Selecting from a Partition: Example" on page 18-28

---

hierarchical_query_clause

The *hierarchical_query_clause* lets you select rows in a hierarchical order.

SELECT statements that contain hierarchical queries can contain the LEVEL pseudocolumn in the select list. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, and so on. The number of levels returned by a hierarchical query may be limited by available user memory.

If you specify this clause, do not specify either ORDER BY or GROUP BY, as they will destroy the hierarchical order of the CONNECT BY results. If you want to order rows of siblings of the same parent, use the ORDER SIBLINGS BY clause.

> **See Also:** "Hierarchical Queries" on page 8-3 for a discussion of hierarchical queries and "Using the LEVEL Pseudocolumn: Examples" on page 18-40

START WITH Clause

Specify a condition that identifies the row(s) to be used as the root(s) of a hierarchical query. Oracle uses as root(s) all rows that satisfy this condition. If you omit this clause, then Oracle uses all rows in the table as root rows. The START WITH condition can contain a subquery, but it cannot contain a scalar subquery expression.

CONNECT BY Clause

Specify a condition that identifies the relationship between parent rows and child rows of the hierarchy. The *connect_by_condition* can be any condition as described in Chapter 5, "Conditions". However, it must use the PRIOR operator to refer to the parent row.

Restriction on the CONNECT BY clause: The *connect_by_condition* cannot contain a regular subquery or a scalar subquery expression.

See Also:

- "Pseudocolumns" on page 2-83 for more information on LEVEL
- "Hierarchical Queries" on page 8-3 for general information on hierarchical queries
- "Hierarchical Query Examples" on page 18-32

Notes on hierarchical queries:

If you specify a hierarchical query and also specify the ORDER BY clause, then the ORDER BY clause takes precedence over any ordering specified by the hierarchical query, unless you specify the SIBLINGS keyword in the ORDER BY clause.

The manner in which Oracle processes a WHERE clause (if any) in a hierarchical query depends on whether the WHERE clause contains a join:

- If the WHERE predicate contains a join, Oracle applies the join predicates *before* doing the CONNECT BY processing.
- If the WHERE clause does not contain a join, Oracle applies all predicates other than the CONNECT BY predicates *after* doing the CONNECT BY processing without affecting the other rows of the hierarchy.

group_by_clause

Specify the GROUP BY clause if you want Oracle to group the selected rows based on the value of *expr*(s) for each row and return a single row of summary information for each group. If this clause contains CUBE or ROLLUP extensions, then Oracle produces superaggregate groupings in addition to the regular groupings.

Expressions in the GROUP BY clause can contain any columns of the tables, views, or materialized views in the FROM clause, regardless of whether the columns appear in the select list.

The GROUP BY clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

See Also:

- *Oracle9i Data Warehousing Guide* for an expanded discussion and examples of using SQL grouping syntax for data aggregation
- the GROUP_ID, GROUPING, and GROUPING_ID functions on page 6-69 for examples
- "Using the GROUP BY Clause: Examples" on page 18-30

ROLLUP The ROLLUP operation in the *simple_grouping_clause* groups the selected rows based on the values of the first n, n-1, n-2, ... 0 expressions in the GROUP BY specification, and returns a single row of summary for each group. You can use the ROLLUP operation to produce **subtotal values** by using it with the SUM function. When used with SUM, ROLLUP generates subtotals from the most detailed level to the grand total. Aggregate functions such as COUNT can be used to produce other kinds of superaggregates.

For example, given three expressions (n=3) in the ROLLUP clause of the *simple_grouping_clause*, the operation results in n+1 = 3+1 = 4 groupings.

Rows grouped on the values of the first 'n' expressions are called **regular rows**, and the others are called **superaggregate rows**.

See Also: *Oracle9i Data Warehousing Guide* for information on using ROLLUP with materialized views

CUBE The CUBE operation in the *simple_grouping_clause* groups the selected rows based on the values of all possible combinations of expressions in the specification, and returns a single row of summary information for each group. You can use the CUBE operation to produce **cross-tabulation values**.

For example, given three expressions (n=3) in the CUBE clause of the *simple_grouping_clause*, the operation results in $2^n = 2^3 = 8$ groupings. Rows grouped on the values of 'n' expressions are called **regular rows**, and the rest are called **superaggregate rows**.

See Also:

- *Oracle9i Data Warehousing Guide* for information on using CUBE with materialized views
- "Using the GROUP BY CUBE Clause: Example" on page 18-30

GROUPING SETS GROUPING SETS are a further extension of the GROUP BY clause that let you specify multiple groupings of data. Doing so facilitates efficient aggregation by pruning the aggregates you do not need. You specify just the desired groups, and Oracle does not need to perform the full set of aggregations generated by CUBE or ROLLUP. Oracle computes all groupings specified in the GROUPING SETS clause and combines the results of individual groupings with a UNION ALL operation. The UNION ALL means that the result set can include duplicate rows.

Within the GROUP BY clause, you can combine expressions in various ways:

- To specify **composite columns**, you group columns within parentheses so that Oracle treats them as a unit while computing ROLLUP or CUBE operations.
- To specify **concatenated grouping sets**, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that Oracle combines them into a single GROUP BY clause. The result is a cross-product of groupings from each grouping set.

  See Also: "Using the GROUPING SETS Clause: Example" on page 18-31

HAVING Clause

Use the HAVING clause to restrict the groups of returned rows to those groups for which the specified *condition* is TRUE. If you omit this clause, then Oracle returns summary rows for all groups.

Specify GROUP BY and HAVING after the *where_clause* and *hierarchical_query_clause*. If you specify both GROUP BY and HAVING, then they can appear in either order.

Restriction on the HAVING clause: The HAVING condition cannot contain a scalar subquery expression.

See Also: "Using the HAVING Condition: Example" on page 18-33

Restrictions on the GROUP BY clause:

- The expressions can be of any form except scalar subquery expressions.
- You cannot specify LOB columns, nested tables, or varrays as part of *expr*.
- If the *group_by_clause* references any object type columns, then the query will not be parallelized.

> **See Also:** the syntax description of *expr* in "About SQL Expressions" on page 4-2 and the syntax description of *condition* in Chapter 5, "Conditions"

Set Operators: UNION, UNION ALL, INTERSECT, MINUS

These set operators combine the rows returned by two SELECT statements into a single result. The number and datatypes of the columns selected by each component query must be the same, but the column lengths can be different. The names of the columns in the result set are the names of the expressions in the select list preceding the set operator.

If you combine more than two queries with set operators, then Oracle evaluates adjacent queries from left to right. You can use parentheses to specify a different order of evaluation.

> **See Also:** "The UNION [ALL], INTERSECT, MINUS Operators" on page 8-6 for information on these operators

Restrictions on set operators:

- The set operators are not valid on columns of type BLOB, CLOB, BFILE, VARRAY, or nested table.
- The UNION, INTERSECT, and MINUS operators are not valid on LONG columns.
- If the select list preceding the set operator contains an expression,then you must provide a column alias for the expression in order to refer to it in the *order_by_clause*.
- You cannot also specify the *for_update_clause* with these set operators.
- You cannot specify the *order_by_clause* in the *subquery* of these operators.
- You cannot use these operators in SELECT statements containing TABLE collection expressions.

Note: To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

order_by_clause

Use the ORDER BY clause to order rows returned by the statement. Without an *order_by_clause*, no guarantee exists that the same query executed more than once will retrieve rows in the same order.

SIBLINGS The SIBLINGS keyword is valid only if you also specify the *hierarchical_query_clause* (CONNECT BY). ORDER SIBLINGS BY preserves any ordering specified in the hierarchical query clause and then applies the *order_by_clause* to the siblings of the hierarchy.

expr *expr* orders rows based on their value for *expr*. The expression is based on columns in the select list or columns in the tables, views, or materialized views in the FROM clause.

position Specify *position* to order rows based on their value for the expression in this position of the select list; *position* must be an integer.

> **See Also:** "Sorting Query Results" on page 8-9 for a discussion of ordering query results

You can specify multiple expressions in the *order_by_clause*. Oracle first sorts rows based on their values for the first expression. Rows with the same value for the first expression are then sorted based on their values for the second expression, and so on. Oracle sorts nulls following all others in ascending order and preceding all others in descending order.

ASC | DESC Specify whether the ordering sequence is ascending or descending. ASC is the default.

NULLS FIRST | NULLS LAST Specify whether returned rows containing null values should appear first or last in the ordering sequence.

NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.

Restrictions on the *order_by_clause*:

- If you have specified the DISTINCT operator in this statement, then this clause cannot refer to columns unless they appear in the select list.
- An *order_by_clause* can contain no more than 255 expressions.
- You cannot order by a LOB column, nested table, or varray.
- If you specify a *group_by_clause* in the same statement, then this *order_by_clause* is restricted to the following expressions:
  - Constants
  - Aggregate functions
  - Analytic functions
  - The functions USER, UID, and SYSDATE
  - Expressions identical to those in the *group_by_clause*
  - Expressions comprising the preceding expressions that evaluate to the same value for all rows in a group.

See Also: "Using the ORDER BY Clause: Examples" on page 18-33

for_update_clause

The FOR UPDATE clause lets you lock the selected rows so that other users cannot lock or update the rows until you end your transaction. You can specify this clause only in a top-level SELECT statement (not in subqueries).

Note: Prior to updating a LOB value, you must lock the row containing the LOB. One way to lock the row is with an embedded SELECT ... FOR UPDATE statement. You can do this using one of the programmatic languages or DBMS_LOB package. For more information on lock rows before writing to a LOB, see *Oracle9i Application Developer's Guide - Large Objects (LOBs)*.

Nested table rows are not locked as a result of locking the parent table rows. If you want the nested table rows to be locked, then you must lock them explicitly.

Restrictions on the *for_update_clause*:

- You cannot specify this clause with the following other constructs: the DISTINCT or CURSOR operator, set operators, *group_by_clause*, or aggregate functions.
- The tables locked by this clause must all be located on the same database, and on the same database as any LONG columns and sequences referenced in the same statement.

See Also: "Using the FOR UPDATE Clause: Examples" on page 18-33

OF ... *column*

Use the OF ... *column* clause to lock the select rows only for a particular table or view in a join. The columns in the OF clause only indicate which table or view rows are locked. The specific columns that you specify are not significant. However, you must specify an actual column name, not a column alias. If you omit this clause, then Oracle locks the selected rows from all the tables in the query.

NOWAIT | WAIT

The NOWAIT and WAIT clauses let you tell Oracle how to proceed if the SELECT statement attempts to lock a row that is locked by another user.

NOWAIT Specify NOWAIT to return control to you immediately if a lock exists.

WAIT Specify WAIT to instruct Oracle to wait *integer* seconds for the row to become available, and then return control to you.

If you specify neither WAIT nor NOWAIT, then Oracle waits until the row is available and then returns the results of the SELECT statement.

Examples

Subquery Factoring: Example The following statement creates the query names dept_costs and avg_cost for the initial query block containing a join, and then uses the query names in the body of the main query.

```
WITH
   dept_costs AS (
      SELECT department_name, SUM(salary) dept_total
         FROM employees e, departments d
         WHERE e.department_id = d.department_id
```

```
      GROUP BY department_name),
   avg_cost AS (
      SELECT SUM(dept_total)/COUNT(*) avg
      FROM dept_costs)
SELECT * FROM dept_costs
   WHERE dept_total >
      (SELECT avg FROM avg_cost)
      ORDER BY department_name;

DEPARTMENT_NAME                DEPT_TOTAL
------------------------------ ----------
Sales                              313800
Shipping                           156400
```

Simple Query Examples The following statement selects rows from the `employees` table with the department number of 30:

```
SELECT *
   FROM employees
   WHERE department_id = 30;
```

The following statement selects the name, job, salary and department number of all employees except purchase clerks from department number 30:

```
SELECT last_name, job_id, salary department_id
   FROM employees
   WHERE NOT (job_id = 'PU_CLERK' AND department_id = 30);
```

The following statement selects from subqueries in the `FROM` clause and gives departments' total employees and salaries as a decimal value of all the departments:

```
SELECT a.department_id "Department",
   a.num_emp/b.total_count "%_Employees",
   a.sal_sum/b.total_sal "%_Salary"
FROM
(SELECT department_id, COUNT(*) num_emp, SUM(salary) sal_sum
   FROM employees
   GROUP BY department_id) a,
(SELECT COUNT(*) total_count, SUM(salary) total_sal
   FROM employees) b;
```

Selecting from a Partition: Example You can select rows from a single partition of a partitioned table by specifying the keyword `PARTITION` in the `FROM` clause. This

SQL statement assigns an alias for and retrieves rows from the `sales_q2_2000` partition of the sample table `sh.sales`:

```
SELECT * FROM sales PARTITION (sales_q2_2000) s
   WHERE s.amount_sold > 10000;
```

The following example selects rows from the `oe.orders` table for orders earlier than a specified date:

```
SELECT * FROM orders
   WHERE order_date < TO_DATE('1999-06-15', 'YYYY-MM-DD');
```

Selecting a Sample: Examples The following query estimates the number of orders in the `oe.orders` table:

```
SELECT COUNT(*) * 100 FROM orders SAMPLE BLOCK (1);
```

The following example creates a sampled subset of the sample table `hr.employees` table and then joins the resulting sampled table with `departments`. This operation circumvents the restriction that you cannot specify the *`sample_clause`* in join queries:

```
CREATE TABLE sample_emp AS
   SELECT employee_id, department_id FROM employees SAMPLE(10);

SELECT e.employee_id FROM sample_emp e, departments d
   WHERE e.department_id = d.department_id
   AND d.department_name = 'Sales';
```

Using Flashback Queries: Example The following statements show a current value from the sample table `hr.employees` and then changes the value:

```
SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      3800

UPDATE employees SET salary = 4000
   WHERE last_name = 'Chung';
1 row updated.

SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
```

```
----------
      4000
```

To learn what the value was before the update, you can use the following flashback query:

```
SELECT salary FROM employees
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY)
   WHERE last_name = 'Chung';

    SALARY
----------
      3800
```

To revert to the earlier value, use the flashback query as the subquery of another UPDATE statement:

```
UPDATE employees SET salary =
   (SELECT salary FROM employees
   AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY)
   WHERE last_name = 'Chung')
   WHERE last_name = 'Chung';
1 row updated.

SELECT salary FROM employees
   WHERE last_name = 'Chung';

    SALARY
----------
      3800
```

Using the GROUP BY Clause: Examples To return the minimum and maximum salaries for each department in the employees table, issue the following statement:

```
SELECT department_id, MIN(salary), MAX (salary)
     FROM employees
     GROUP BY department_id;
```

To return the minimum and maximum salaries for the clerks in each department, issue the following statement:

```
SELECT department_id, MIN(salary), MAX (salary)
     FROM employees
     WHERE job_id = 'PU_CLERK'
     GROUP BY department_id;
```

Using the GROUP BY CUBE Clause: Example To return the number of employees and their average yearly salary across all possible combinations of department and

job category, issue the following query on the sample tables `hr.employees` and `hr.departments`:

```
SELECT DECODE(GROUPING(department_name), 1, 'All Departments',
      department_name) AS department_name,
   DECODE(GROUPING(job_id), 1, 'All Jobs', job_id) AS job_id,
   COUNT(*) "Total Empl", AVG(salary) * 12 "Average Sal"
   FROM employees e, departments d
   WHERE d.department_id = e.department_id
   GROUP BY CUBE (department_name, job_id);

DEPARTMENT_NAME                JOB_ID     Total Empl Average Sal
------------------------------ ---------- ---------- -----------
Accounting                     AC_ACCOUNT          1       99600
Accounting                     AC_MGR              1      144000
Accounting                     All Jobs            2      121800
Administration                 AD_ASST             1       52800
.
.
.
All Departments                ST_MAN              5       87360
All Departments                All Jobs          107  77798.1308
```

Using the GROUPING SETS Clause: Example The following example finds the sum of sales aggregated for three precisely specified groups:

- `(channel_desc, calendar_month_desc, country_id)`
- `(channel_desc, country_id)`
- `(calendar_month_desc, country_id)`

Without the `GROUPING SETS` syntax, you would have to write less efficient queries with more complicated SQL. For example, you could run three separate queries and `UNION` them, or run a query with a `CUBE(channel_desc, calendar_month_desc, country_id)` operation and filter out 5 of the 8 groups it would generate.

```
SELECT channel_desc, calendar_month_desc, co.country_id,
      TO_CHAR(sum(amount_sold) , '9,999,999,999') SALES$
   FROM sales, customers, times, channels, countries co
   WHERE sales.time_id=times.time_id
      AND sales.cust_id=customers.cust_id
      AND sales.channel_id= channels.channel_id
      AND customers.country_id = co.country_id
      AND channels.channel_desc IN ('Direct Sales', 'Internet')
      AND times.calendar_month_desc IN ('2000-09', '2000-10')
```

```
      AND co.country_id IN ('UK', 'US')
  GROUP BY GROUPING SETS(
      (channel_desc, calendar_month_desc, co.country_id),
      (channel_desc, co.country_id),
      ( calendar_month_desc, co.country_id) );

CHANNEL_DESC         CALENDAR CO SALES$
-------------------- -------- -- --------------
Direct Sales         2000-09  UK      1,378,126
Direct Sales         2000-10  UK      1,388,051
Direct Sales         2000-09  US      2,835,557
Direct Sales         2000-10  US      2,908,706
Internet             2000-09  UK        911,739
Internet             2000-10  UK        876,571
Internet             2000-09  US      1,732,240
Internet             2000-10  US      1,893,753
Direct Sales                  UK      2,766,177
Direct Sales                  US      5,744,263
Internet                      UK      1,788,310
Internet                      US      3,625,993
                     2000-09  UK      2,289,865
                     2000-09  US      4,567,797
                     2000-10  UK      2,264,622
                     2000-10  US      4,802,459
```

See Also: the functions GROUP_ID, GROUPING, and GROUPING_ID on page 6-69 for more information on those functions

Hierarchical Query Examples The following query with a CONNECT BY clause defines a hierarchical relationship in which the employee_id value of the parent row is equal to the manager_id value of the child row:

```
SELECT last_name, employee_id, manager_id FROM employees
   CONNECT BY employee_id = manager_id;
```

In the following CONNECT BY clause, the PRIOR operator applies only to the employee_id value. To evaluate this condition, Oracle evaluates employee_id values for the parent row and manager_id, salary, and commission_pct values for the child row:

```
SELECT last_name, employee_id, manager_id FROM employees
   CONNECT BY PRIOR employee_id = manager_id
   AND salary > commission_pct;
```

To qualify as a child row, a row must have a manager_id value equal to the employee_id value of the parent row and it must have a salary value greater than its commission_pct value.

Using the HAVING Condition: Example To return the minimum and maximum salaries for the employees in each department whose lowest salary is less than $5,000, issue the next statement:

```
SELECT department_id, MIN(salary), MAX (salary)
   FROM employees
   GROUP BY department_id
   HAVING MIN(salary) < 5000;

DEPARTMENT_ID MIN(SALARY) MAX(SALARY)
------------- ----------- -----------
           10        4400        4400
           30        2500       11000
           50        2100        8200
           60        4200        9000
```

Using the ORDER BY Clause: Examples To select all salesmen's records from employees, and order the results by commission in descending order, issue the following statement:

```
SELECT *
   FROM employees
   WHERE job_id = 'PU_CLERK'
   ORDER BY commission_pct DESC;
```

To select information from employees ordered first by ascending department number and then by descending salary, issue the following statement:

```
SELECT last_name, department_id, salary
   FROM employees
   ORDER BY department_id ASC, salary DESC;
```

To select the same information as the previous SELECT and use the positional ORDER BY notation, issue the following statement:

```
SELECT last_name, department_id, salary
   FROM employees
   ORDER BY 2 ASC, 3 DESC;
```

Using the FOR UPDATE Clause: Examples The following statement locks rows in the employees table with purchasing clerks located in Oxford (location_id

2500) and locks rows in the departments table with departments in Oxford that have purchasing clerks:

```
SELECT e.employee_id, e.salary, e.commission_pct
   FROM employees e, departments d
   WHERE job_id = 'SA_REP'
   AND e.department_id = d.department_id
   AND location_id = 2500
   FOR UPDATE;
```

The following statement locks only those rows in the employees table with purchasing clerks located in Oxford (location_id 2500). No rows are locked in the departments table:

```
SELECT e.employee_id, e.salary, e.commission_pct
   FROM employees e, departments d
   WHERE job_id = 'SA_REP'
   AND e.department_id = d.department_id
   AND location_id = 2500
   FOR UPDATE OF e.salary;
```

Using the WITH CHECK OPTION Clause: Example The following statement is legal even though the third value inserted violates the condition of the subquery *where_clause*:

```
INSERT INTO (SELECT department_id, department_name, location_id
   FROM departments WHERE location_id < 2000)
   VALUES (9999, 'Entertainment', 2500);
```

However, the following statement is illegal because it contains the WITH CHECK OPTION clause:

```
INSERT INTO (SELECT department_id, department_name, location_id
   FROM departments WHERE location_id < 2000 WITH CHECK OPTION)
   VALUES (9999, 'Entertainment', 2500);
     *
ERROR at line 2:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Using Join Queries: Examples The following examples show various ways of joining tables in a query. In the first example, an equijoin returns the name and job of each employee and the number and name of the department in which the employee works:

```
SELECT last_name, job_id, departments.department_id, department_name
   FROM employees, departments
```

```
   WHERE employees.department_id = departments.department_id;

LAST_NAME           JOB_ID     DEPARTMENT_ID DEPARTMENT_NAME
------------------- ---------- ------------- ----------------------
...
Sciarra             FI_ACCOUNT           100 Finance
Urman               FI_ACCOUNT           100 Finance
Popp                FI_ACCOUNT           100 Finance
...
```

You must use a join to return this data because employee names and jobs are stored in a different table than department names. Oracle combines rows of the two tables according to this join condition:

```
employees.department_id = departments.department_id
```

The following equijoin returns the name, job, department number, and department name of all sales managers:

```
SELECT last_name, job_id, departments.department_id, department_name
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   AND job_id = 'SA_MAN';

LAST_NAME           JOB_ID     DEPARTMENT_ID DEPARTMENT_NAME
------------------- ---------- ------------- ----------------------
Russell             SA_MAN                80 Sales
Partners            SA_MAN                80 Sales
Errazuriz           SA_MAN                80 Sales
Cambrault           SA_MAN                80 Sales
Zlotkey             SA_MAN                80 Sales
```

This query is identical to the preceding example, except that it uses an additional *where_clause* condition to return only rows with a `job` value of 'SA_MAN'.

Using Subqueries: Examples To determine who works in the same department as employee '`Lorentz`', issue the following statement:

```
SELECT last_name, department_id FROM employees
   WHERE department_id =
     (SELECT department_id FROM employees
      WHERE last_name = 'Lorentz');
```

To give all employees in the `employees` table a 10% raise if they have changed jobs (that is, if they appear in the `job_history` table), issue the following statement:

```
UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id IN (SELECT employee_id FROM job_history);
```

To create a second version of the `departments` table `new_departments`, with only three of the columns of the original table, issue the following statement:

```
CREATE TABLE new_departments
   (department_id, department_name, location_id)
   AS SELECT department_id, department_name, location_id
   FROM departments;
```

Using Self Joins: Example The following query uses a self join to return the name of each employee along with the name of the employee's manager. (A `WHERE` clause is added to shorten the output.)

```
SELECT e1.last_name||' works for '||e2.last_name
   "Employees and Their Managers"
   FROM employees e1, employees e2
   WHERE e1.manager_id = e2.employee_id
      AND e1.last_name LIKE 'R%';

Employees and Their Managers
------------------------------
Rajs works for Mourgos
Raphaely works for King
Rogers works for Kaufling
Russell works for King
```

The join condition for this query uses the aliases `e1` and `e2` for the sample table `employees`:

```
e1.manager_id = e2.employee_id
```

Using Outer Joins: Examples The following example uses a left outer join to return the names of all departments in the sample schema `hr`, even if no employees have been assigned to the departments:

```
SELECT d.department_id, e.last_name
   FROM departments d LEFT OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id;

DEPARTMENT_ID LAST_NAME
------------- -------------------------
           10 Whalen
```

```
           20 Hartstein
           20 Fay
           30 Raphaely
...
          250
          260
          270
```

Users familiar with the traditional Oracle outer joins syntax will recognize the same query in this form:

```
SELECT d.department_id, e.last_name
   FROM departments d, employees e
   WHERE d.department_id = e.department_id(+)
   ORDER BY d.department_id;
```

Oracle Corporation strongly recommends that you use the more flexible Oracle9i FROM clause join syntax shown in the former example.

The left outer join returns all departments, including those without any employees. The same statement with a right outer join returns all employees, including those not yet assigned to a department:

Note: The employee Zeuss was added to the employees table for these examples, and is not part of the sample data.

```
SELECT d.department_id, e.last_name
   FROM departments d RIGHT OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id;

DEPARTMENT_ID LAST_NAME
------------- -------------------------
...
          110 Higgins
          110 Gietz
              Grant
              Zeuss
```

It is not clear from this result whether employees Grant and Zeuss have department_id NULL, or whether their department_id is not in the departments table. To determine this requires a full outer join:

```
SELECT d.department_id as d_dept_id, e.department_id as e_dept_id,
```

```
      e.last_name
   FROM departments d FULL OUTER JOIN employees e
   ON d.department_id = e.department_id
   ORDER BY d.department_id;

 D_DEPT_ID  E_DEPT_ID LAST_NAME
---------- ---------- -------------------------
  ...
       110        110 Gietz
       110        110 Higgins
  ...
       260
       270
                  999 Zeuss
                      Grant
```

Because the column names in this example are the same in both tables in the join, you can also use the common column feature (the `USING` clause) of the join syntax, which coalesces the two matching columns `department_id`. The output is the same as for the preceding example:

```
SELECT department_id AS d_e_dept_id, e.last_name
   FROM departments d FULL OUTER JOIN employees e
   USING (department_id)
   ORDER BY department_id;

D_E_DEPT_ID LAST_NAME
----------- -------------------------
  ...
        110 Higgins
        110 Gietz
  ...
        260
        270
            Grant
            Zeuss
```

Table Collections: Examples You can perform DML operations on nested tables only if they are defined as columns of a table. Therefore, when the *query_table_expr_clause* of an `INSERT`, `DELETE`, or `UPDATE` statement is a *table_collection_expression*, the collection expression must be a subquery that selects the table's nested table column. The examples that follow are based on this scenario:

```
CREATE TYPE ProjectType AS OBJECT(
```

```
    pno    NUMBER,
    pname  CHAR(31),
    budget NUMBER);
CREATE TYPE ProjectSet AS TABLE OF ProjectType;

CREATE TABLE dept_work (dno NUMBER, dname CHAR(31), projs
ProjectSet)
    NESTED TABLE projs STORE AS
        ProjectSetTable ((Primary Key(Nested_Table_Id, pno))
ORGANIZATION
INDEX COMPRESS 1);

INSERT INTO dept_work VALUES (1, 'Engineering', ProjectSet());
```

This example inserts into the 'Engineering' department's 'projs' nested table:

```
INSERT INTO TABLE(SELECT d.projs
                  FROM   dept_work d
                  WHERE  d.dno = 1)
  VALUES (1, 'Collection Enhancements', 10000);
```

This example updates the 'Engineering' department's 'projs' nested table:

```
UPDATE TABLE(SELECT d.projs
             FROM   dept_work d
             WHERE  d.dno = 1) p
  SET  p.budget = p.budget + 1000;
```

This example deletes from the 'Engineering' department's 'projs' nested table

```
DELETE TABLE(SELECT d.projs
             FROM   dept_work d
             WHERE  d.dno = 1) p
 WHERE p.budget > 100000;
```

Collection Unnesting: Examples Suppose the database contains a table hr_info with columns dept, location, and mgr, and a column of nested table type people which has name, dept, and sal columns. You could get all the rows from hr_info and all the rows from people using the following statement:

```
SELECT t1.dept, t2.* FROM hr_info t1, TABLE(t1.people) t2
   WHERE t2.dept = t1.dept;
```

Now suppose that people is not a nested table column of hr_info, but is instead a separate table with columns name, dept, address, hiredate, and sal. You can extract the same rows as in the preceding example with this statement:

```
SELECT t1.department, t2.*
   FROM hr_info t1, TABLE(CAST(MULTISET(
      SELECT t3.name, t3.dept, t3.sal FROM people t3
      WHERE t3.dept = t1.dept)
      AS NESTED_PEOPLE)) t2;
```

Finally, suppose that `people` is neither a nested table column of table `hr_info` nor a table itself. Instead, you have created a function `people_func` that extracts from various sources the name, department, and salary of all employees. You can get the same information as in the preceding examples with the following query:

```
SELECT t1.dept, t2.* FROM HY_INFO t1, TABLE(CAST
   (people_func( ... ) AS NESTED_PEOPLE)) t2;
```

> **See Also:** *Oracle9i Application Developer's Guide - Fundamentals* for more examples of collection unnesting.

Using the LEVEL Pseudocolumn: Examples The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is 'AD_VP'. The child rows of a parent row are defined to be those who have the employee number of the parent row as their manager number.

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
        employee_id, manager_id, job_id
    FROM employees
    START WITH job_id = 'AD_VP'
    CONNECT BY PRIOR employee_id = manager_id;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
Kochhar                    101        100 AD_VP
  Greenberg                108        101 FI_MGR
    Faviet                 109        108 FI_ACCOUNT
    Chen                   110        108 FI_ACCOUNT
    Sciarra                111        108 FI_ACCOUNT
    Urman                  112        108 FI_ACCOUNT
    Popp                   113        108 FI_ACCOUNT
  Whalen                   200        101 AD_ASST
  Mavris                   203        101 HR_REP
  Baer                     204        101 PR_REP
  Higgins                  205        101 AC_MGR
    Gietz                  206        205 AC_ACCOUNT
De Haan                    102        100 AD_VP
  Hunold                   103        102 IT_PROG
```

```
    Ernst                   104        103 IT_PROG
    Austin                  105        103 IT_PROG
    Pataballa               106        103 IT_PROG
    Lorentz                 107        103 IT_PROG
```

The following statement is similar to the previous one, except that it does not select employees with the job 'FI_MAN'.

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
        employee_id, manager_id, job_id
    FROM employees
    WHERE job_id != 'FI_MGR'
    START WITH job_id = 'AD_VP'
    CONNECT BY PRIOR employee_id = manager_id;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
Kochhar                    101        100 AD_VP
    Faviet                 109        108 FI_ACCOUNT
    Chen                   110        108 FI_ACCOUNT
    Sciarra                111        108 FI_ACCOUNT
    Urman                  112        108 FI_ACCOUNT
    Popp                   113        108 FI_ACCOUNT
  Whalen                   200        101 AD_ASST
  Mavris                   203        101 HR_REP
  Baer                     204        101 PR_REP
  Higgins                  205        101 AC_MGR
    Gietz                  206        205 AC_ACCOUNT
De Haan                    102        100 AD_VP
  Hunold                   103        102 IT_PROG
    Ernst                  104        103 IT_PROG
    Austin                 105        103 IT_PROG
    Pataballa              106        103 IT_PROG
    Lorentz                107        103 IT_PROG
```

Oracle does not return the manager greenberg, although it does return employees who are managed by greenberg.

The following statement is similar to the first one, except that it uses the LEVEL pseudocolumn to select only the first two levels of the management hierarchy:

```
SELECT LPAD(' ',2*(LEVEL-1)) || last_name org_chart,
employee_id, manager_id, job_id
    FROM employees
    START WITH job_id = 'AD_PRES'
```

```
    CONNECT BY PRIOR employee_id = manager_id AND LEVEL <= 2;

ORG_CHART          EMPLOYEE_ID MANAGER_ID JOB_ID
------------------ ----------- ---------- ----------
King                       100            AD_PRES
  Kochhar                  101        100 AD_VP
  De Haan                  102        100 AD_VP
  Raphaely                 114        100 PU_MAN
  Weiss                    120        100 ST_MAN
  Fripp                    121        100 ST_MAN
  Kaufling                 122        100 ST_MAN
  Vollman                  123        100 ST_MAN
  Mourgos                  124        100 ST_MAN
  Russell                  145        100 SA_MAN
  Partners                 146        100 SA_MAN
  Errazuriz                147        100 SA_MAN
  Cambrault                148        100 SA_MAN
  Zlotkey                  149        100 SA_MAN
  Hartstein                201        100 MK_MAN
```

Using Distributed Queries: Example This example shows a query that joins the `departments` table on the local database with the `employees` table on the `houston` database:

```
SELECT last_name, department_name
   FROM employees@houston, departments
   WHERE employees.department_id = departments.department_id;
```

Using Correlated Subqueries: Examples The following examples show the general syntax of a correlated subquery:

```
SELECT select_list
    FROM table1 t_alias1
    WHERE expr operator
        (SELECT column_list
            FROM table2 t_alias2
            WHERE t_alias1.column
               operator t_alias2.column);

UPDATE table1 t_alias1
    SET column =
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
```

```
DELETE FROM table1 t_alias1
    WHERE column operator
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
```

The following statement returns data about employees whose salaries exceed their department average. The following statement assigns an alias to `employees`, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT department_id, last_name, salary
   FROM employees x
   WHERE salary > (SELECT AVG(salary)
      FROM employees
      WHERE x.department_id = department_id)
   ORDER BY department_id;
```

For each row of the `employees` table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs the following steps for each row of the `employees` table:

1. The `department_id` of the row is determined.
2. The `department_id` is then used to evaluate the parent query.
3. If that row's salary is greater than the average salary for that row's department, then the row is returned.

The subquery is evaluated once for each row of the `employees` table.

Selecting from the DUAL Table: Example The following statement returns the current date:

```
SELECT SYSDATE FROM DUAL;
```

You could select SYSDATE from the `employees` table, but Oracle would return 14 rows of the same SYSDATE, one for every row of the `employees` table. Selecting from DUAL is more convenient.

Selecting Sequence Values: Examples The following statement increments the `employees_seq` sequence and returns the new value:

```
SELECT employees_seq.nextval
    FROM dual;
```

The following statement selects the current value of `employees_seq`:

```
SELECT employees_seq.currval
    FROM dual;
```

SET CONSTRAINT[S]

Purpose

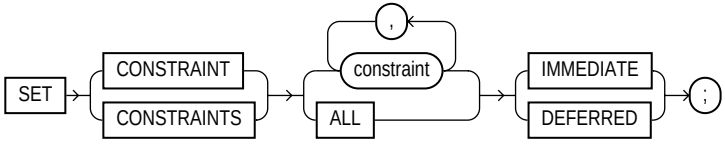
Use the SET CONSTRAINTS statement to specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement or when the transaction is committed.

Prerequisites

To specify when a deferrable constraint is checked, you must have SELECT privilege on the table to which the constraint is applied unless the table is in your schema.

Syntax

set_constraints::=



Keywords and Parameters

constraint

Specify the name of one or more integrity constraints.

ALL

Specify ALL to set all deferrable constraints for this transaction.

IMMEDIATE

Specify IMMEDIATE to indicate that the conditions specified by the deferrable constraint are checked immediately after each DML statement.

DEFERRED

Specify DEFERRED to indicate that the conditions specified by the deferrable constraint are checked when the transaction is committed.

Note: You can verify the success of deferrable constraints prior to committing them by issuing a `SET CONSTRAINTS ALL IMMEDIATE` statement.

Examples

Setting Constraints: Examples The following statement sets all deferrable constraints in this transaction to be checked immediately following each DML statement:

```
SET CONSTRAINTS ALL IMMEDIATE;
```

The following statement checks three deferred constraints when the transaction is committed:

```
SET CONSTRAINTS emp_job_nn, emp_salary_min,
   hr.emp_job_fk@houston DEFERRED;
```

SET ROLE

Purpose

Use the SET ROLE statement to enable and disable roles for your current session.

When a user logs on, Oracle enables all privileges granted explicitly to the user and all privileges in the user's default roles. During the session, the user or an application can use the SET ROLE statement any number of times to change the roles currently enabled for the session. The number of roles that can be concurrently enabled is limited by the initialization parameter MAX_ENABLED_ROLES.

You can see which roles are currently enabled by examining the SESSION_ROLES data dictionary view.

See Also:

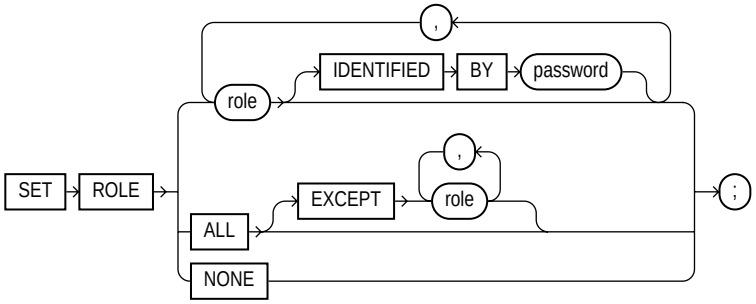
- CREATE ROLE on page 14-77 for information on creating roles
- ALTER USER on page 12-21 for information on changing a user's default roles
- *Oracle9i Database Reference* for information on the SESSION_ROLES session parameter

Prerequisites

You must already have been granted the roles that you name in the SET ROLE statement.

Syntax

set_role::=



Keywords and Parameters

role

Specify a role to be enabled for the current session. Any roles not listed and not already enabled are disabled for the current session.

In the IDENTIFIED BY *password* clause, specify the password for a role. If the role has a password, then you must specify the password to enable the role.

Restriction on setting roles: You cannot specify a role unless it was granted to you either directly or through other roles.

ALL Clause

Specify ALL to enable all roles granted to you for the current session except those optionally listed in the EXCEPT clause.

Roles listed in the EXCEPT clause must be roles granted directly to you. They cannot be roles granted to you through other roles.

If you list a role in the EXCEPT clause that has been granted to you both directly and through another role, then the role remains enabled by virtue of the role to which it has been granted.

Restriction on the ALL clause: You cannot use this clause to enable roles with passwords that have been granted directly to you.

NONE

Specify NONE to disable all roles for the current session, including the DEFAULT role.

Examples

Setting Roles: Examples To enable the role gardener identified by the password marigolds for your current session, issue the following statement:

```
SET ROLE gardener IDENTIFIED BY marigolds;
```

To enable all roles granted to you for the current session, issue the following statement:

```
SET ROLE ALL;
```

To enable all roles granted to you except dw_manager, issue the following statement:

```
SET ROLE ALL EXCEPT dw_manager;
```

To disable all roles granted to you for the current session, issue the following statement:

```
SET ROLE NONE;
```

SET TRANSACTION

Purpose

Use the SET TRANSACTION statement to establish the current transaction as read only or read write, establish its isolation level, or assign it to a specified rollback segment.

The operations performed by a SET TRANSACTION statement affect only your current transaction, not other users or other transactions. Your transaction ends whenever you issue a COMMIT or ROLLBACK statement. Oracle implicitly commits the current transaction before and after executing a data definition language (DDL) statement.

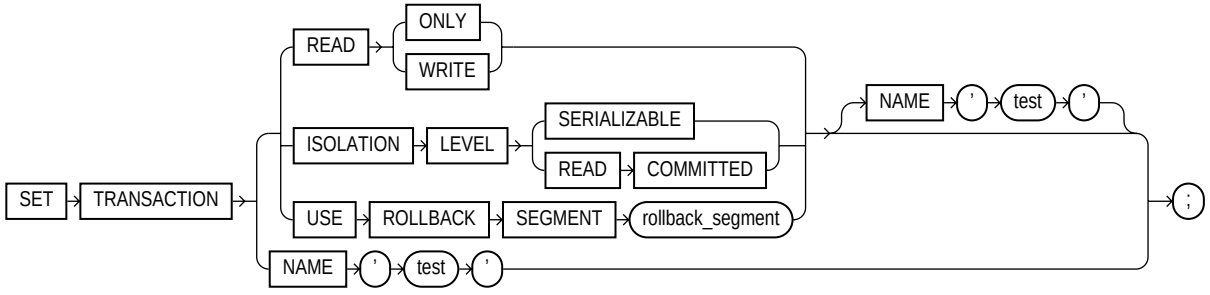
> **See Also:** COMMIT on page 12-72 and ROLLBACK on page 17-100

Prerequisites

If you use a SET TRANSACTION statement, then it must be the first statement in your transaction. However, a transaction need not have a SET TRANSACTION statement.

Syntax

set_transaction::=



Keywords and Parameters

READ ONLY

The READ ONLY clause establishes the current transaction as a read-only transaction. This clause established **transaction-level read consistency**.

All subsequent queries in that transaction only see changes committed before the transaction began. Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

Note: This clause is not supported for the user SYS. That is, queries by SYS will return changes made during the transaction even if SYS has set the transaction to be READ ONLY.

Restriction on read-only transactions: Only the following statements are permitted in a read-only transaction:

- Subqueries (that is, SELECT statements without the *for_update_clause*)
- LOCK TABLE
- SET ROLE
- ALTER SESSION
- ALTER SYSTEM

See Also: *Oracle9i Database Concepts*

READ WRITE

Specify READ WRITE to establish the current transaction as a read/write transaction. This clause establishes **statement-level read consistency**, which is the default.

Restriction on read/write transactions: You cannot toggle between transaction-level and statement-level read consistency in the same transaction.

ISOLATION LEVEL Clause

Use the ISOLATION LEVEL clause to specify how transactions containing database modifications are handled.

- The SERIALIAZBLE setting specifies serializable transaction isolation mode as defined in the SQL92 standard. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may

have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.

> **Note:** The COMPATIBLE initialization parameter must be set to 7.3.0 or higher for SERIALIZABLE mode to work.

- The READ COMMITTED setting is the default Oracle transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.

USE ROLLBACK SEGMENT Clause

Specify USE ROLLBACK SEGMENT to assign the current transaction to the specified rollback segment. This clause also implicitly establishes the transaction as a read / write transaction.

This clause lets you to assign transactions of different types to rollback segments of different sizes. For example:

- If no long-running queries are concurrently reading the same tables, then you can assign small transactions to small rollback segments, which are more likely to remain in memory.
- You can assign transactions that modify tables that are concurrently being read by long-running queries to large rollback segments, so that the rollback information needed for the read-consistent queries is not overwritten.
- You can assign transactions that insert, update, or delete large amounts of data to rollback segments large enough to hold the rollback information for the transaction.

You cannot use the READ ONLY clause and the USE ROLLBACK SEGMENT clause in a single SET TRANSACTION statement or in different statements in the same transaction. Read-only transactions do not generate rollback information and therefore are not assigned rollback segments.

NAME Clause

Use the NAME clause to assign a name to the current transaction. This clause is especially useful in distributed database environments when you must identify and resolve in-doubt transactions. The *text* string is limited to 255 bytes.

If you specify a name for a distributed transaction, then when the transaction commits, the name becomes the commit comment, overriding any comment specified explicitly in the COMMIT statement.

Examples

Setting Transactions: Examples The following statements could be run at midnight of the last day of every month to count the products and quantities on hand in the Toronto warehouse in the sample Order Entry (oe) schema. This report would not be affected by any other user who might be adding or removing inventory to a different warehouse.

```
COMMIT;

SET TRANSACTION READ ONLY NAME 'Toronto';

SELECT product_id, quantity_on_hand FROM inventories
   WHERE warehouse_id = 5;

COMMIT;
```

The first COMMIT statement ensures that SET TRANSACTION is the first statement in the transaction. The last COMMIT statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

The following statement assigns your current transaction to the rollback segment rs_one:

```
SET TRANSACTION USE ROLLBACK SEGMENT rs_one;
```

TRUNCATE

Caution: You cannot roll back a TRUNCATE statement.

Purpose

Use the TRUNCATE statement to remove all rows from a table or cluster. By default, Oracle also deallocates all space used by the removed rows except that specified by the MINEXTENTS storage parameter and sets the NEXT storage parameter to the size of the last extent removed from the segment by the truncation process.

Removing rows with the TRUNCATE statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table invalidates the table's dependent objects, requires you to regrant object privileges on the table, and requires you to re-create the table's indexes, integrity constraints, and triggers and respecify its storage parameters. Truncating has none of these effects.

See Also:

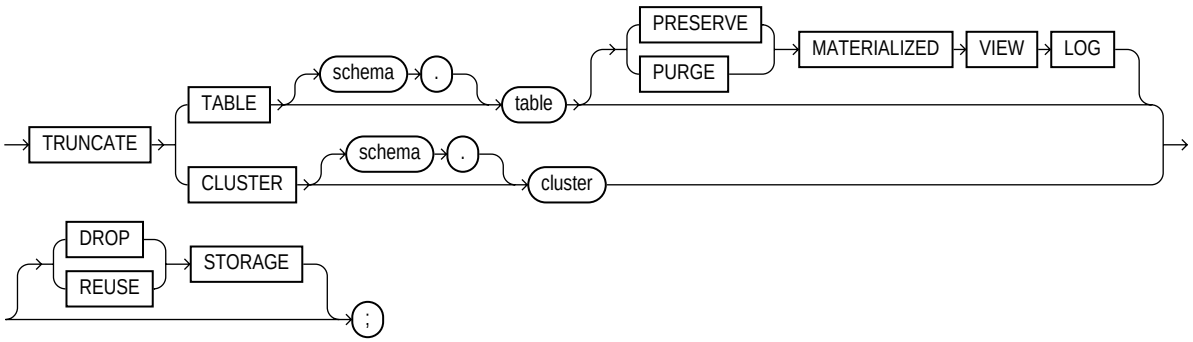
- DELETE on page 16-55 and DROP TABLE on page 17-6 for information on other ways to drop table data from the database
- DROP CLUSTER on page 16-67 for information on dropping cluster tables

Prerequisites

To truncate a table or cluster, the table or cluster must be in your schema or you must have DROP ANY TABLE system privilege.

Syntax

truncate::=



Keywords and Parameters

TABLE Clause

Specify the schema and name of the table to be truncated. This table cannot be part of a cluster. If you omit *schema*, then Oracle assumes the table is in your own cluster.

- You can truncate index-organized tables and temporary tables. When you truncate a temporary table, only the rows created during the current session are removed.
- Oracle changes the NEXT storage parameter of *table* to be the size of the last extent deleted from the segment in the process of truncation.
- Oracle also automatically truncates and resets any existing UNUSABLE indicators for the following indexes on *table*: range and hash partitions of local indexes and subpartitions of local indexes.
- If *table* is not empty, then Oracle marks UNUSABLE all nonpartitioned indexes and all partitions of global partitioned indexes on the table.
- For a domain index, this statement invokes the appropriate truncate routine to truncate the domain index data.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for more information on domain indexes

- If *table* (whether it is a regular or index-organized table) contains LOB columns, then all LOB data and LOB index segments are truncated.
- If *table* is partitioned, then all partitions or subpartitions, as well as the LOB data and LOB index segments for each partition or subpartition, are truncated.

> **Note:** When you truncate a table, Oracle automatically removes all data in the table's indexes and any materialized view direct-path INSERT information held in association with the table. (This information is independent of any materialized view log.) If this direct-path INSERT information is removed, then an incremental refresh of the materialized view may lose data.

Restrictions on truncating tables:

- You cannot individually truncate a table that is part of a cluster. You must either truncate the cluster, delete all rows from the table, or drop and re-create the table.
- You cannot truncate the parent table of an enabled referential integrity constraint. You must disable the constraint before truncating the table. (An exception is that you may truncate the table if the integrity constraint is self-referential.)
- If *table* belongs to a hierarchy, then it must be the root of the hierarchy.
- If a domain index is defined on *table*, then neither the index nor any index partitions can be marked IN_PROGRESS.

MATERIALIZED VIEW LOG Clause

The MATERIALIZED VIEW LOG clause lets you specify whether a materialized view log defined on the table is to be preserved or purged when the table is truncated. This clause permits materialized view master tables to be reorganized through export/import without affecting the ability of primary-key materialized views defined on the master to be fast refreshed. To support continued fast refresh of primary-key materialized views, the materialized view log must record primary-key information.

> **Note:** The keyword SNAPSHOT is supported in place of MATERIALIZED VIEW for backward compatibility.

PRESERVE Specify PRESERVE if any materialized view log should be preserved when the master table is truncated. This is the default.

PURGE Specify PURGE if any materialized view log should be purged when the master table is truncated.

> **See Also:** *Oracle9i Replication* for more information about materialized view logs and the TRUNCATE statement

CLUSTER Clause

Specify the schema and name of the cluster to be truncated. You can truncate only an indexed cluster, not a hash cluster. If you omit *schema*, then Oracle assumes the cluster is in your own schema.

When you truncate a cluster, Oracle also automatically deletes all data in the indexes of the cluster tables.

STORAGE Clauses

The STORAGE clauses let you determine what happens to the space freed by the truncated rows. The DROP STORAGE clause and REUSE STORAGE clause also apply to the space freed by the data deleted from associated indexes.

DROP STORAGE Specify DROP STORAGE to deallocate all space from the deleted rows from the table or cluster except the space allocated by the MINEXTENTS parameter of the table or cluster. This space can subsequently be used by other objects in the tablespace. Oracle also sets the NEXT storage parameter to the size of the last extent removed from the segment in the truncation process. This is the default.

REUSE STORAGE Specify REUSE STORAGE to retain the space from the deleted rows allocated to the table or cluster. Storage values are not reset to the values when the table or cluster was created. This space can subsequently be used only by new data in the table or cluster resulting from insert or update operations. This clause leaves storage parameters at their current settings.

Note: If you have specified more than one free list for the object you are truncating, then the REUSE STORAGE clause also removes any mapping of free lists to instances and resets the high-water mark to the beginning of the first extent.

Examples

Truncating a Table: Example The following statement removes all rows from a copy of the sample table `hr.employees` and returns the freed space to the tablespace containing `employees`:

```
TRUNCATE TABLE employees_demo;
```

The preceding statement also removes all data from all indexes on `employees` and returns the freed space to the tablespaces containing them.

Retaining Free Space After Truncate: Example The following statement removes all rows from all tables in the `personnel` cluster, but leaves the freed space allocated to the tables:

```
TRUNCATE CLUSTER personnel REUSE STORAGE
```

The preceding statement also removes all data from all indexes on the tables in the `personnel` cluster.

Preserving Materialized View Logs After Truncate: Example The following statements are examples of truncate statements that preserve materialized view logs:

```
TRUNCATE TABLE sales_demo PRESERVE MATERIALIZED VIEW LOG;
TRUNCATE TABLE orders_demo;
```

UPDATE

Purpose

Use the UPDATE statement to change existing values in a table or in a view's base table.

Prerequisites

For you to update values in a table, the table must be in your own schema or you must have UPDATE privilege on the table.

For you to update values in the base table of a view:

- You must have UPDATE privilege on the view, and
- Whoever owns the schema containing the view must have UPDATE privilege on the base table.

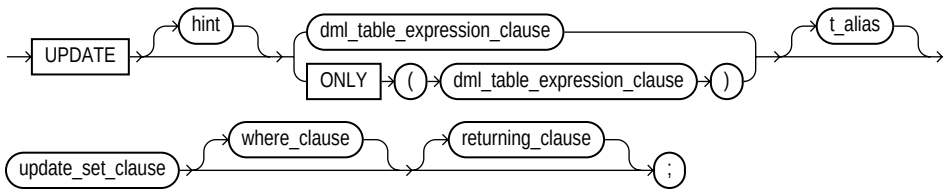
The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

You must also have the SELECT privilege on the object you want to update if:

- The object is on a remote database or
- The SQL92_SECURITY initialization parameter is set to TRUE and the UPDATE operation references table columns (such as the columns in a *where_clause*).

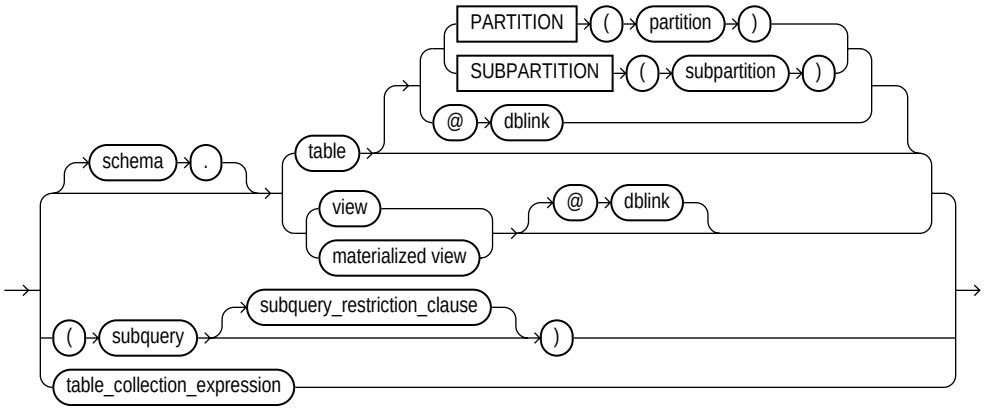
Syntax

update::=



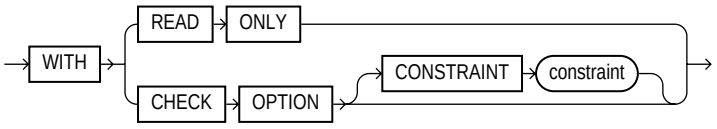
(*DML_table_expression_clause::=* on page 18-60, *update_set_clause::=* on page 18-61, *where_clause::=* on page 18-61, *returning_clause::=* on page 18-61)

DML_table_expression_clause::=



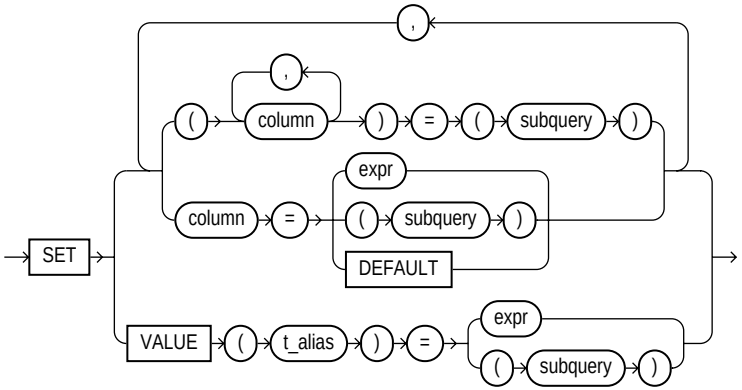
(*subquery::=* on page 18-5—part of SELECT syntax, *subquery_restriction_clause::=* on page 18-60, *table_collection_expression::=* on page 18-60)

subquery_restriction_clause::=

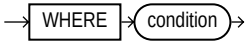


table_collection_expression::=

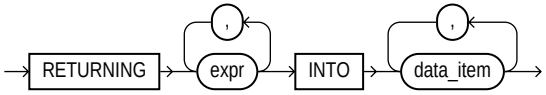
update_set_clause::=



where_clause::=



returning_clause::=



Keywords and Parameters

hint

Specify a comment that passes instructions to the optimizer on choosing an execution plan for the statement.

You can place a parallel hint immediately after the UPDATE keyword to parallelize both the underlying scan and UPDATE operations.

See Also:

- *Oracle9i Database Performance Tuning Guide and Reference* and "Hints" on page 2-92 for the syntax and description of hints
- *Oracle9i Database Performance Tuning Guide and Reference* and *Oracle9i Database Concepts* for detailed information about parallel DML

DML_table_expression_clause

The ONLY clause applies only to views. Specify ONLY syntax if the view in the UPDATE clause is a view that belongs to a hierarchy and you do not want to update rows from any of its subviews.

> **See Also:** "Restrictions on the dml_table_expression_clause:" on page 18-64 and "Updating a Table: Examples" on page 18-68

schema

Specify the schema containing the table or view. If you omit *schema*, then Oracle assumes the table or view is in your own schema.

table | *view* | *subquery*

Specify the name of the table or view, or the columns returned by a subquery, to be updated. Issuing an UPDATE statement against a table fires any UPDATE triggers associated with the table. If you specify *view*, then Oracle updates the view's base table.

If *table* (or the base table of *view*) contains one or more domain index columns, then this statement executes the appropriate indextype update routine.

> **See Also:** *Oracle9i Data Cartridge Developer's Guide* for more information on these routines

PARTITION (*partition*) | SUBPARTITION (*subpartition*)

Specify the name of the partition or subpartition within *table* targeted for updates. You need not specify the partition name when updating values in a partitioned table. However in some cases specifying the partition name can be more efficient than a complicated *where_clause*.

> **See Also:** "Updating a Partition: Example" on page 18-69

dblink

Specify a complete or partial name of a database link to a remote database where the table or view is located. You can use a database link to update a remote table or view only if you are using Oracle's distributed functionality.

If you omit *dblink*, then Oracle assumes the table or view is on the local database.

> **See Also:** "Referring to Objects in Remote Databases" on page 2-118 for information on referring to database links

subquery_restriction_clause

Use the *subquery_restriction_clause* to restrict the subquery in one of the following ways:

WITH READ ONLY Specify WITH READ ONLY to indicate that the table or view cannot be updated.

WITH CHECK OPTION Specify WITH CHECK OPTION to indicate that Oracle prohibits any changes to the table or view that would produce rows that are not included in the subquery.

CONSTRAINT *constraint* Specify the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

> **See Also:** "Using the WITH CHECK OPTION Clause: Example" on page 18-34

table_collection_expression

The *table_collection_expression* lets you inform Oracle that the value of *collection_expression* should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a function, or a collection constructor. Regardless of its form, it must return a collection value (that is, a value whose type is nested table or varray). This process of extracting the elements of a collection is called **collection unnesting**.

> **Note:** In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expression* was expressed as "THE subquery". That usage is now deprecated.

You can use a *table_collection_expression* to update rows in one table based on rows from another table. For example, you could roll up four quarterly sales tables into a yearly sales table.

t_alias

Specify a **correlation name** (alias) for the table, view, or subquery to be referenced elsewhere in the statement.

Note: This alias is *required* if the *dml_table_expression_clause* references any object type attributes or object type methods.

See Also: "Correlated Update: Example" on page 18-69

Restrictions on the *dml_table_expression_clause*:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked IN_PROGRESS or FAILED.
- You cannot insert into a partition if any affected index partitions are marked UNUSABLE.
- You cannot specify the *order_by_clause* in the subquery of the *dml_table_expression_clause*.
- You cannot update a view except with INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate or analytic function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list
  - Joins (with some exceptions). See *Oracle9i Database Administrator's Guide* for details.
- If a view was created with the WITH CHECK OPTION, then you can update the view only if the resulting data satisfies the view's defining query.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, then the UPDATE statement will fail unless the SKIP_UNUSABLE_INDEXES session parameter has been set to TRUE.

See Also: ALTER SESSION on page 10-2 for information on the SKIP_UNUSABLE_INDEXES session parameter

update_set_clause

The *update_set_clause* lets you set column values.

column

Specify the name of a column of the table or view that is to be updated. If you omit a column of the table from the *update_set_clause*, then that column's value remains unchanged.

If *column* refers to a LOB object attribute, then you must first initialize it with a value of empty or null. You cannot update it with a literal. Also, if you are updating a LOB value using some method other than a direct UPDATE SQL statement, then you must first lock the row containing the LOB.

> **See Also:** *for_update_clause* on page 18-26

If *column* is part of the partitioning key of a partitioned table, then UPDATE will fail if you change a value in the column that would move the row to a different partition or subpartition, unless you enable row movement.

> **See Also:** the *row_movement_clause* of CREATE TABLE on page 15-7 or ALTER TABLE on page 11-2

In addition, if *column* is part of the partitioning key of a list-partitioned table, then UPDATE will fail if you specify a value for the column that does not already exist in the *partition_value* list of one of the partitions.

subquery

Specify a subquery that returns exactly one row for each row updated.

- If you specify only one column in the *update_set_clause*, then the subquery can return only one value.
- If you specify multiple columns in the *update_set_clause*, then the subquery must return as many values as you have specified columns.

You can use the *flashback_clause* of within the subquery to update *table* with past data.

> **See Also:** the *flashback_clause* of SELECT on page 18-14 for more information on this clause

If the subquery returns no rows, then the column is assigned a null.

Note: If this *subquery* refers to remote objects, then the UPDATE operation can run in parallel as long as the reference does not loop back to an object on the local database. However, if the *subquery* in the *dml_table_expression_clause* refers to any remote objects, then the UPDATE operation will run serially without notification.

See Also:

- SELECT on page 18-4 and "Using Subqueries" on page 8-13
- *parallel_clause* of CREATE TABLE on page 15-56

expr

Specify an expression that resolves to the new value assigned to the corresponding column.

See Also: Chapter 4, "Expressions" for the syntax of *expr* and "Using the SET VALUE Clause: Example" on page 18-69

DEFAULT Specify DEFAULT to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, then Oracle sets the column to null.

Restriction on setting default values: You cannot specify DEFAULT if you are updating a view.

VALUE Clause

The VALUE clause lets you specify the entire row of an object table.

Restriction on the VALUE clause: You can specify this clause only for an object table.

Note: If you insert string literals into a RAW column, during subsequent queries, then Oracle will perform a full table scan rather than using any index that might exist on the RAW column.

See Also: "Using the SET VALUE Clause: Example" on page 18-69

where_clause

The *where_clause* lets you restrict the rows updated to those for which the specified *condition* is true. If you omit this clause, then Oracle updates all rows in the table or view.

The *where_clause* determines the rows in which values are updated. If you do not specify the *where_clause*, then all rows are updated. For each row that satisfies the *where_clause*, the columns to the left of the equals (=) operator in the *update_set_clause* are set to the values of the corresponding expressions on the right. The expressions are evaluated as the row is updated.

See Also: Chapter 5, "Conditions" for the syntax of *condition*

returning_clause

The returning clause retrieves the rows affected by a DML (INSERT, UPDATE, or DELETE) statement. You can specify this clause for tables and materialized views, and for views with a single base table.

When operating on a single row, a DML statement with a *returning_clause* can retrieve column expressions using the affected row, rowid, and REFs to the affected row and store them in host variables or PL/SQL variables.

When operating on multiple rows, a DML statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the affected rows in bind arrays.

expr Each item in the *expr* list must be a valid expression syntax. All forms are valid except scalar subquery expressions.

INTO The INTO clause indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.

data_item Each *data_item* is a host variable or PL/SQL variable that stores the retrieved *expr* value.

For each expression in the RETURNING list, you must specify a corresponding type-compatible PL/SQL variable or host variable in the INTO list.

Restrictions:

- You cannot specify the *returning_clause* for a multitable insert.
- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.

- You cannot specify this clause for a view on which an INSTEAD OF trigger has been defined.

  See Also: *PL/SQL User's Guide and Reference* for information on using the BULK COLLECT clause to return multiple values to collection variables

Examples

Updating a Table: Examples The following statement gives null commissions to all employees with the job SA_CLERK:

```
UPDATE employees
   SET commission_pct = NULL
   WHERE job_id = 'SA_CLERK';
```

The following statement promotes Douglas Grant to manager of Department 20 with a $1,000 raise:

```
UPDATE employees SET
    job_id = 'SA_MAN', salary = salary + 1000, department_id = 120
    WHERE first_name||' '||last_name = 'Douglas Grant';
```

The following statement increases the balance of bank account number 5001 in the accounts table on a remote database accessible through the database link boston:

```
UPDATE finance.accounts@boston
    SET balance = balance + 500
    WHERE acc_no = 5001;
```

The next example shows the following syntactic constructs of the UPDATE statement:

- Both forms of the *update_set_clause* together in a single statement
- A correlated subquery
- A *where_clause* to limit the updated rows

```
UPDATE employees a
    SET department_id =
        (SELECT department_id
            FROM departments
            WHERE location_id = '2100'),
        (salary, commission_pct) =
        (SELECT 1.1*AVG(salary), 1.5*AVG(commission_pct)
```

```
        FROM employees b
        WHERE a.department_id = b.department_id)
  WHERE department_id IN
      (SELECT department_id
        FROM departments
        WHERE location_id = 2900
            OR location_id = 2700);
```

The preceding UPDATE statement performs the following operations:

- Updates only those employees who work in Geneva or Munich (locations 2900 and 2700)
- Sets `department_id` for these employees to the `department_id` corresponding to Bombay (`location_id` 2100)
- Sets each employee's salary to 1.1 times the average salary of their department
- Sets each employee's commission to 1.5 times the average commission of their department

Updating a Partition: Example The following example updates values in a single partition of the `sales` table:

```
UPDATE sales PARTITION (sales_q1_1999) s
   SET s.promo_id = 494
   WHERE amount_sold > 9000;
```

Using the SET VALUE Clause: Example The following statement updates a row of object table `table1` by selecting a row from another object table `table2`:

```
UPDATE table1 p SET VALUE(p) =
   (SELECT VALUE(q) FROM table2 q WHERE p.id = q.id)
   WHERE p.id = 10;
```

The subquery uses the `value` object reference function in its expression.

Correlated Update: Example The following example updates particular rows of the `projs` nested table corresponding to the department whose department equals 123:

```
UPDATE TABLE(SELECT projs
           FROM dept d WHERE d.dno = 123)  p
  SET p.budgets = p.budgets + 1
  WHERE p.pno IN (123, 456);
```

Using the RETURNING Clause During UPDATE: Example The following example returns values from the updated row and stores the result in PL/SQL variables bnd1, bnd2, bnd3:

```
UPDATE employees
  SET job_id ='SA_MAN', salary = salary + 1000, department_id = 140
  WHERE last_name = 'Jones'
  RETURNING salary*0.25, last_name, department_id
    INTO :bnd1, :bnd2, :bnd3;
```

A

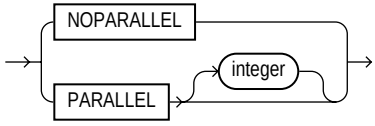
How to Read Syntax Diagrams

Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

Commands and other keywords appear in UPPERCASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lowercase inside ovals. Variables are used for the parameters. Punctuation, operators, delimiters, and terminators appear inside circles.

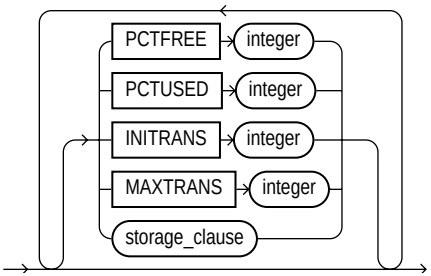
If the syntax diagram has more than one path, you can choose any path to travel. For example, in the following syntax you can specify either NOPARALLEL or PARALLEL:

parallel_clause::=



If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list. For example, in the following syntax diagram, you can specify one or more of the five parameters in the stack:

physical_attributes_clause::=



The following table shows parameters that appear in the syntax diagrams and provides examples of the values you might substitute for them in your statements:

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter. For a list of all types of objects, see the section, "Schema Objects" on page 2-107.	`emp`
c	The substitution value must be a single character from your database character set.	`T` `s`
'text'	The substitution value must be a text string in single quotes. See the syntax description of '*text*' in "Text Literals" on page 2-54.	`'Employee records'`
char	The substitution value must be an expression of datatype CHAR or VARCHAR2 or a character literal in single quotes.	`ename` `'Smith'`
condition	The substitution value must be a condition that evaluates to TRUE or FALSE. See the syntax description of *condition* in Chapter 5, "Conditions".	`ename >'A'`

date
d
	The substitution value must be a date constant or an expression of DATE datatype.	`TO_DATE(`
		`'01-Jan-1994',`
		`'DD-MON-YYYY')`

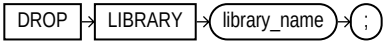
expr	The substitution value can be an expression of any datatype as defined in the syntax description of *expr* in "About SQL Expressions" on page 4-2.	`sal + 1000`

Parameter	Description	Examples
integer	The substitution value must be an integer as defined by the syntax description of integer in "Integer Literals" on page 2-55.	`72`
number *m* *n*	The substitution value must be an expression of `NUMBER` datatype or a number constant as defined in the syntax description of *number* in "Number Literals" on page 2-56.	`AVG(sal)` `15 * 7`
raw	The substitution value must be an expression of datatype `RAW`.	`HEXTORAW('7D')`
subquery	The substitution value must be a `SELECT` statement that will be used in another SQL statement. See SELECT on page 18-4.	`SELECT ename` `FROM emp`
db_name	The substitution value must be the name of a nondefault database in an embedded SQL program.	`sales_db`
db_string	The substitution value must be the database identification string for an Oracle Net database connection. For details, see the user's guide for your specific Oracle Net protocol.	—

Required Keywords and Parameters

Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *library_name* is a required parameter:

drop_library::=



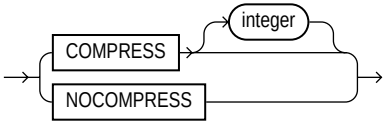
If there is a library named `HQ_LIB`, then, according to the diagram, the following statement is valid:

```
DROP LIBRARY hq_lib;
```

If multiple keywords or parameters appear in a vertical list that intersects the main path, one of them is required. That is, you must choose one of the keywords or

parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the two settings:

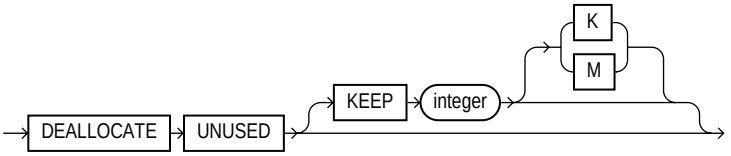
key_compression::=



Optional Keywords and Parameters

If keywords and parameters appear in a vertical list *above* the main path, they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:

deallocate_unused_clause::=



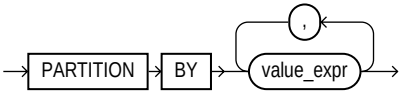
According to the diagram, all of the following statements are valid:

```
DEALLOCATE UNUSED;
DEALLOCATE UNUSED KEEP 1000;
DEALLOCATE UNUSED KEEP 10M;
```

Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, after choosing one value expression, you can go back repeatedly to choose another, separated by commas.

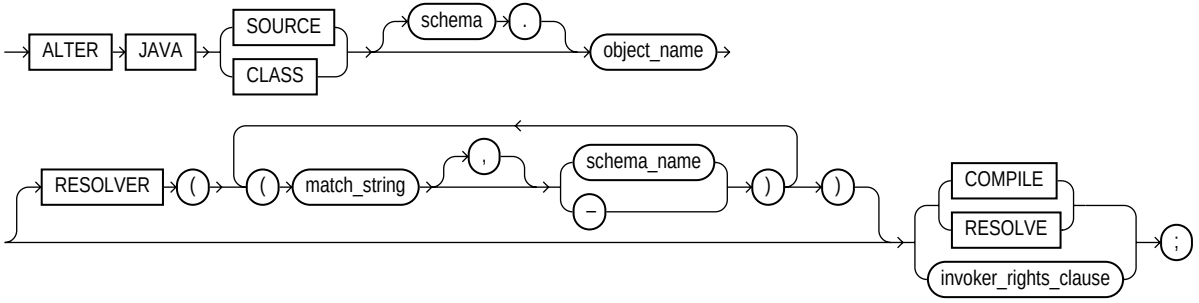
query_partition_clause::=



Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end to end. The following example is a two-part diagram:

alter_java::=



According to the diagram, the following statement is valid:

```
ALTER JAVA SOURCE jsource_1 COMPILE;
```

Database Objects

The names of Oracle identifiers, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an Oracle identifier is enclosed by double quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle identifiers are not case sensitive except when enclosed by double quotation marks.

For more information, see "Schema Object Naming Rules" on page 2-111.

B

Oracle and Standard SQL

This appendix discusses Oracle's conformance with the SQL:1999 standards. The mandatory portion of SQL:1999 is known as Core SQL:1999 and is found in SQL:1999 Part 2 (Foundation) and Part 5 (Bindings). The Foundation features are analyzed in Annex F of Part 2 in the table "SQL/Foundation feature taxonomy and definition for Core SQL". The Bindings features are analyzed in Annex F of Part 5 in the table "SQL/Bindings feature taxonomy and definition for Core SQL".

This appendix declares Oracle's conformance to the SQL standards established by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). (The ANSI and ISO SQL standards are identical.) It contains the following sections:

- ANSI Standards
- ISO Standards
- Oracle Compliance
- FIPS Compliance

ANSI Standards

The following documents of the American National Standards Institute (ANSI) relate to SQL:

- ANSI / ISO / IEC 9075-1:1999, Information technology—Database languages—SQL—Part 1: Framework (SQL / Framework)
  - ANSI / ISO / IEC 9075-1:1999 / Amd.1:2000
- ANSI / ISO / IEC 9075-2:1999, Information technology—Database languages—SQL—Part 2: Foundation (SQL / Foundation)
- ANSI / ISO / IEC 9075-5:1999, Information technology—Database languages—SQL—Part 5: Host Language Bindings (SQL / Bindings)

You can obtain a copy of ANSI standards from this address:

American National Standards Institute
11 West 42nd Street
New York, NY 10036 USA
Telephone: +1.212.642.4900
Fax: +1.212.398.0023

or from their Web site:

`http://webstore.ansi.org/ansidocstore/default.asp`

A subset of ANSI standards, including the SQL standard, are X3 or NCITS standards. You can obtain these from the National Committee for Information Technology Standards (NCITS) at:

`http://www.cssinfo.com/ncitsgate.html`

ISO Standards

The following documents of the International Organization for Standardization (ISO) relate to SQL:

- ISO / IEC 9075-1:1999, Information technology—Database languages—SQL—Part 1: Framework (SQL / Framework)
  - ISO / IEC 9075-1:1999 / Amd.1:2000
- ISO / IEC 9075-2:1999, Information technology—Database languages—SQL—Part 2: Foundation (SQL / Foundation)

- ISO / IEC 9075-5:1999, Information technology—Database languages—SQL—Part 5: Host Language Bindings (SQL / Bindings)

You can obtain a copy of ISO standards from this address:

International Organization for Standardization
1 Rue de Varembé
Case postale 56
CH-1211, Geneva 20, Switzerland
Phone: +41.22.749.0111
Fax: +41.22.733.3430
Web site: http: / / www.iso.ch /

or from their web store:

`http://www.iso.ch/cate/cat.html`

Oracle Compliance

The ANSI and ISO SQL standards require conformance claims to state the type of conformance and the implemented facilities. The following products provide full or partial conformance with the ANSI and ISO standards as described in the tables that follow:

- Oracle9*i* database server
- Pro*C / C++, release 9.2.0
- Pro*COBOL, release 9.2.0
- Pro*Fortran, release 1.8.77
- SQL Module for Ada (Mod*Ada), release 9.2.0
- Pro*COBOL 1.8, release 1.8.77
- Pro*PL / I, release 1.6.28
- OTT, release 9.2.0.
- OTT8, release 8.1.8

The Core SQL:1999 features that Oracle fully supports are listed in Table B–1:

Table B–1 Fully Supported Core SQL:1999 Features

Feature ID	Feature
E011	Numeric data types

Table B–1 (Cont.) Fully Supported Core SQL:1999 Features

Feature ID	Feature
E031	Identifiers
E061	Basic predicates and search conditions
E081	Basic privileges
E091	Set functions
E101	Basic data manipulation
E111	Single row `SELECT` statement
E131	Null value support (nulls in lieu of values)
E141	Basic integrity constraints
E151	Transaction support
E152	Basic `SET TRANSACTION` statement
E153	Updatable queries with subqueries
E161	SQL comments using leading double minus
E171	`SQLSTATE` support
F041	Basic joined table
F051	Basic date and time
F081	`UNION` and `EXCEPT` in views
F131	Grouped operations
F181	Multiple module support
F201	`CAST` function
F221	Explicit defaults
F261	`CASE` expressions
F311	Schema definition statement
F471	Scalar subquery values
F481	Expanded `NULL` predicate
B011	Embedded Ada
B012	Embedded C
B013	Embedded COBOL

Table B–1 (Cont.) Fully Supported Core SQL:1999 Features

Feature ID	Feature
B014	Embedded Fortran
T431	Extended grouping capabilities
T611	Elementary OLAP operators
T621	Enhanced numeric functions

The Core SQL:1999 features that Oracle partially supports are listed in Table B–2:

Table B–2 Partially Supported Core SQL:1999 Features

Feature ID, Feature	Partial Support
E021, Character data types	Oracle fully supports these subfeatures: ■ E021-01, `CHARACTER` data type ■ E021-07, Character concatenation ■ E021-08, `UPPER` and `LOWER` functions ■ E021-09, `TRIM` function ■ E021-10, Implicit casting among character data types ■ E021-11, Character comparison Oracle partially supports these subfeatures: ■ E021-02, `CHARACTER VARYING` data type (Oracle does not distinguish a zero-length `VARCHAR` string from `NULL`) ■ E021-03, Character literals (Oracle regards the zero-length literal `''` as being null) Oracle has equivalent functionality for these subfeatures: ■ E021-04, `CHARACTER_LENGTH` function: use `LENGTH` function instead ■ E021-05, `OCTET_LENGTH` function: use `LENGTHB` function instead ■ E021-06, `SUBSTRING` function: use `SUBSTR` function instead ■ E021-11, `POSITION` function: use `INSTR` function instead

Table B–2 (Cont.) Partially Supported Core SQL:1999 Features

Feature ID, Feature	Partial Support
F031, Basic schema manipulation	Oracle fully supports these subfeatures: ■ F031-01, CREATE TABLE statement to create persistent base tables ■ F031-02, CREATE VIEW statement ■ F031-03, GRANT statement Oracle partially supports this subfeature: ■ F031-04, ALTER TABLE statement: ADD COLUMN clause (Oracle does not support the optional keyword COLUMN in this syntax) Oracle does not support these subfeatures (because Oracle does not support the keyword RESTRICT): ■ F031-13, DROP TABLE statement: RESTRICT clause ■ F031-16, DROP VIEW statement: RESTRICT clause ■ F031-19, REVOKE statement: RESTRICT clause
E051, Basic query specification	Oracle fully supports the following subfeatures: ■ E051-01, SELECT DISTINCT ■ E051-02, GROUP BY clause ■ E051-04, GROUP BY can contain columns not in <select list> ■ E051-05, Select list items can be renamed ■ E051-06, HAVING clause ■ E051-07, Qualified * in select list Oracle partially supports the following subfeatures: ■ E051-08, Correlation names in FROM clause (Oracle supports correlation names, but not the optional AS keyword) Oracle does not support the following subfeature: ■ E051-09, Rename columns in the FROM clause

Table B–2 (Cont.) Partially Supported Core SQL:1999 Features

Feature ID, Feature	Partial Support
E071, Basic query expressions	Oracle fully supports the following subfeatures: ■ E071-01, `UNION DISTINCT` table operator ■ E071-02, `UNION ALL` able operator ■ E071-05, Columns combined by table operators need not have exactly the same type ■ E071-06, table operators in subqueries Oracle has equivalent functionality for the following subfeature: ■ E071-03, `EXCEPT DISTINCT` table operator: Use `MINUS` instead of `EXCEPT DISTINCT`
E121, Basic cursor support	Oracle fully supports the following subfeatures: ■ E121-01, `DECLARE CURSOR` ■ E121-02, `ORDER BY` columns need not be in select list ■ E121-03, Value expressions in `ORDER BY` clause ■ E121-04, `OPEN` statement ■ E121-06, Positioned `UPDATE` statement ■ E121-07, Positioned `DELETE` statement ■ E121-08, `CLOSE` statement ■ E121-10, `FETCH` statement, implicit `NEXT` Oracle does not support the following subfeatures: ■ E121-17, `WITH HOLD` cursors (in the standard, a cursor is not held through a `ROLLBACK`, but Oracle does hold through `ROLLBACK`)
F812, Basic flagging	Oracle has a flagger, but it flags SQL-92 compliance rather than SQL:1999 compliance

Table B–2 (Cont.) Partially Supported Core SQL:1999 Features

Feature ID, Feature	Partial Support
T321, Basic SQL-invoked routines	Oracle fully supports these subfeatures: ■ T321-03, function invocation ■ T321-04, `CALL` statement Oracle supports these subfeatures with syntactic differences: ■ T321-01, user-defined functions with no overloading ■ T321-02, user-defined procedures with no overloading The Oracle syntax for `CREATE FUNCTION` and `CREATE PROCEDURE` differs from the standard as follows: ■ In the standard, the mode of a parameter (`IN`, `OUT` or `INOUT`) comes before the parameter name, whereas in Oracle it comes after the parameter name. ■ The standard uses `INOUT`, whereas Oracle uses `IN OUT`. ■ Oracle requires either `IS` or `AS` after the return type and before the definition of the routine body, while the standard lacks these keywords. ■ If the routine body is in C (for example), then the standard uses the keywords `LANGUAGE C EXTERNAL NAME` to name the routine, whereas Oracle uses `LANGUAGE C NAME`. ■ If the routine body is in SQL, then Oracle uses its proprietary procedural extension called PL/SQL. Oracle supports the following subfeatures in PL/SQL but not in Oracle SQL: ■ T321-05, `RETURN` statement
T612, Advanced OLAP operators	Oracle fully supports the following subfeatures: ■ T612.b: `WIDTH_BUCKET` function ■ T612.c.ii: `PERCENT_RANK` and `CUME_DIST` functions ■ T612.f.i: Hypothetical set functions and inverse distribution functions

Oracle has equivalent functionality for the features listed in Table B–3:

Table B–3 Equivalent Functionality for Core SQL:1999 Features

Feature ID, Feature	Equivalent Functionality
F021, Basic information schema	Oracle does not have any of the views in this feature. However, Oracle makes the same information available in other metadata views: ■ Instead of TABLES, use ALL_TABLES. ■ Instead of COLUMNS, use ALL_TAB_COLUMNS. ■ Instead of VIEWS, use ALL_VIEWS. However, Oracle's ALL_VIEWS does not display whether a user view was defined WITH CHECK OPTION or if it is updatable. To see whether a view has WITH CHECK OPTION, use ALL_CONSTRAINTS, with TABLE_NAME equal to the view name and look for CONSTRAINT_TYPE equal to 'V'. ■ Instead of TABLE_CONSTRAINTS, REFERENTIAL_CONSTRAINTS and CHECK_CONSTRAINTS, use ALL_CONSTRAINTS. However, Oracle's ALL_CONSTRAINTS does not display whether a constraint is deferrable or initially deferred.

The Core SQL:1999 features that Oracle does not support are listed in Table B–4:

Table B–4 Unsupported Core SQL:1999 Features

Feature ID	Feature
F501	Features and conformance views
S011	Distinct data types

Note: Oracle does not support E182, Module language. Although this feature is listed in Table 31 in the standard, it merely indicates that Core consists of a choice between Module language and embedded language. Module language and embedded language are completely equivalent in capability, differing only in the manner in which SQL statements are associated with the host programming language. Oracle supports embedded language.

FIPS Compliance

Oracle complied fully with last Federal Information Processing Standard (FIPS), which was FIPS PUB 127-2. That standard is no longer published. However, for users whose applications depend on information about the sizes of some database constructs that were defined in FIPS 127-2, we list the details of our compliance in Table B–5.

Table B–5 Sizing for Database Constructs

Database Constructs	FIPS	Oracle9*i*
Length of an identifier (in bytes)	18	30
Length of `CHARACTER` datatype (in bytes)	240	2000
Decimal precision of `NUMERIC` datatype	15	38
Decimal precision of `DECIMAL` datatype	15	38
Decimal precision of `INTEGER` datatype	9	38
Decimal precision of `SMALLINT` datatype	4	38
Binary precision of `FLOAT` datatype	20	126
Binary precision of `REAL` datatype	20	63
Binary precision of `DOUBLE PRECISION` datatype	30	126
Columns in a table	100	1000
Values in an `INSERT` statement	100	1000
SET clauses in an `UPDATE` statement[(a)]	20	1000
Length of a row[(b,c)]	2,000	2,000,000
Columns in a `UNIQUE` constraint	6	32
Length of a `UNIQUE` constraint[(b)]	120	(d)
Length of foreign key column list[(b)]	120	(d)
Columns in a `GROUP BY` clause	6	255[(e)]
Length of `GROUP BY` column list	120	(e)
Sort specifications in `ORDER BY` clause	6	255[(e)]
Length of `ORDER BY` column list	120	(e)
Columns in a referential integrity constraint	6	32

Table B–5 (Cont.) Sizing for Database Constructs

Database Constructs	FIPS	Oracle9*i*
Tables referenced in a SQL statement	15	No limit
Cursors simultaneously open	10	(f)
Items in a SELECT list	100	1000

Notes to Table B–5:

(a) The number of SET clauses in an UPDATE statement refers to the number items separated by commas following the SET keyword.

(b) The FIPS PUB defines the length of a collection of columns to be the sum of: twice the number of columns, the length of each character column in bytes, decimal precision plus 1 of each exact numeric column, binary precision divided by 4 plus 1 of each approximate numeric column.

(c) The Oracle limit for the maximum row length is based on the maximum length of a row containing a LONG value of length 2 gigabytes and 999 VARCHAR2 values, each of length 4000 bytes: 2(254) + 231 + (999(4000)).

(d) The Oracle limit for a UNIQUE key is half the size of an Oracle data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

(e) Oracle places no limit on the number of columns in a GROUP BY clause or the number of sort specifications in an ORDER BY clause. However, the sum of the sizes of all the expressions in either a GROUP BY clause or an ORDER BY clause is limited to the size of an Oracle data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

(f) The Oracle limit for the number of cursors simultaneously opened is specified by the initialization parameter OPEN_CURSORS. The maximum value of this parameter depends on the memory available on your operating system and exceeds 100 in all cases.

Oracle Extensions to Standard SQL

Oracle supports numerous features that extend beyond standard SQL. In your Oracle applications, you can use these extensions just as you can use Core SQL:1999.

If you are concerned with the portability of your applications to other implementations of SQL, use Oracle's FIPS Flagger to help identify the use of Oracle extensions to Entry SQL92 in your embedded SQL programs. The FIPS Flagger is part of the Oracle precompilers and the SQL*Module compiler.

> **See Also:** *Pro*COBOL Precompiler Programmer's Guide* and *Pro*C/C++ Precompiler Programmer's Guide* for information on how to use the FIPS Flagger.

Character Set Support

Oracle supports most national, international, and vendor-specific encoded character set standards. A complete list of character sets supported by Oracle Appears in Appendix A, "Locale Data", in *Oracle9i Database Globalization Support Guide*.

Unicode is a universal encoded character set that lets you store information from any language using a single character set. Unicode is required by modern standards such as XML, Java, JavaScript, and LDAP. Unicode is compliant with ISO/IEC standard 10646. You can obtain a copy of ISO/IEC standard 10646 from this address:

International Organization for Standardization
1 Rue de Varembé
Case postale 56
CH-1211, Geneva 20, Switzerland
Phone: +41.22.749.0111
Fax: +41.22.733.3430
Web site: http://www.iso.ch/

Oracle9*i* complies fully with Unicode 3.0, the third and most recent version of the Unicode standard. For up-to-date information on this standard, visit the Web site of the Unicode Consortium:

`http://www.unicode.org`

Oracle uses UTF-8 (8-bit) encoding by way of three database character sets, two for ASCII-based platforms (UTF8 and AL32UTF8) and one for EBCDIC platforms (UTFE). If you prefer to implement Unicode support incrementally, you can store Unicode data in either the UTF-16 or UTF-8 encoding form, in the national character set, for the SQL `NCHAR` datatypes (`NCHAR`, `NVARCHAR2`, and `NCLOB`).

> **See Also:** *Oracle9i Database Globalization Support Guide* for details on Oracle character set support.

C

Oracle Reserved Words

This appendix lists Oracle reserved words. Words followed by an asterisk (*) are also ANSI reserved words.

> **Note:** In addition to the following reserved words, Oracle uses system-generated names beginning with "SYS_" for implicitly generated schema objects and subobjects. Oracle discourages you from using this prefix in the names you explicitly provide to your schema objects and subobjects to avoid possible conflict in name resolution.

Table C–1 Oracle Reserved Words

ACCESS	CHAR *	DEFAULT *
ADD *	CHECK *	DELETE *
ALL *	CLUSTER	DESC *
ALTER *	COLUMN	DISTINCT *
AND *	COMMENT	DROP *
ANY *	COMPRESS	ELSE *
AS *	CONNECT *	EXCLUSIVE
ASC *	CREATE *	EXISTS
AUDIT	CURRENT *	FILE
BETWEEN *	DATE *	FLOAT *
BY *	DECIMAL *	FOR *

Table C–1 (Cont.) Oracle Reserved Words

FROM *	NOT *	SET *
GRANT *	NOWAIT	SHARE
GROUP *	NULL *	SIZE *
HAVING *	NUMBER	SMALLINT *
IDENTIFIED	OF *	START
IMMEDIATE *	OFFLINE	SUCCESSFUL
IN *	ON *	SYNONYM
INCREMENT	ONLINE	SYSDATE
INDEX	OPTION *	TABLE *
INITIAL	OR *	THEN *
INSERT *	ORDER *	TO *
INTEGER *	PCTFREE	TRIGGER
INTERSECT *	PRIOR *	UID
INTO *	PRIVILEGES *	UNION *
IS *	PUBLIC *	UNIQUE *
LEVEL *	RAW	UPDATE *
LIKE *	RENAME	USER *
LOCK	RESOURCE	VALIDATE
LONG	REVOKE *	VALUES *
MAXEXTENTS	ROW	VARCHAR *
MINUS	ROWID	VARCHAR2
MODE	ROWNUM	VIEW *
MODIFY	ROWS *	WHENEVER *
NOAUDIT	SELECT *	WHERE
NOCOMPRESS	SESSION *	WITH *

D

Examples

The body of the *SQL Reference* contains examples for almost every reference topic. This appendix contains lengthy examples that are appropriate in the context of a single SQL statement. These examples are intended to provide uninterrupted the series of steps that you would use to take advantage of particular Oracle functionality. They do not replace the syntax diagrams and semantics found for each individual SQL statement in the body of the reference. Please use the cross-reference provided to access additional information, such as privileges required and restrictions, as well as syntax.

This appendix contains the following sections:

Using Extensible Indexing

This section provides examples of the steps entailed in a simple but realistic extensible indexing scenario.

Suppose you want to rank the salaries in the HR.employees table and then find those that rank between 10 and 20. You could use the DENSE_RANK function, as follows:

```
SELECT last_name, salary FROM
   (SELECT last_name, DENSE_RANK() OVER
      (ORDER BY salary DESC) rank_val, salary FROM employees)
   WHERE rank_val BETWEEN 10 AND 20;
```

See Also: DENSE_RANK on page 6-53

This nested query is somewhat complex, and it requires a full scan of the employees table as well as a sort. An alternative would be to use extensible indexing to achieve the same goal. The resulting query will be simpler. The query will require only an index scan and a table access by rowid, and will therefore perform much more efficiently.

The first step is to create the implementation type position_im, including method headers for index definition, maintenance, and creation. (Most of the type body uses PL/SQL, which is shown in italics.)

See Also:

- CREATE TYPE on page 16-3 and CREATE TYPE BODY on page 16-25
- *Oracle9i Data Cartridge Developer's Guide* for complete information on the ODCI routines in this statement
- *PL/SQL User's Guide and Reference*

```
CREATE OR REPLACE TYPE position_im AS OBJECT
(
  curnum  NUMBER,
  howmany NUMBER,
  lower_bound NUMBER,  --lower_bound and upper_bound are used for the
  upper_bound NUMBER,  --index-based functional implementation.
  STATIC FUNCTION ODCIGETINTERFACES(ifclist OUT SYS.ODCIOBJECTLIST) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXCREATE
    (ia SYS.ODCIINDEXINFO, parms VARCHAR2, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXTRUNCATE (ia SYS.ODCIINDEXINFO,
```

```
                                    env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXDROP(ia SYS.ODCIINDEXINFO,
                                env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXINSERT(ia SYS.ODCIINDEXINFO, rid ROWID,
                                  newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXDELETE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXUPDATE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  STATIC FUNCTION ODCIINDEXSTART(SCTX IN OUT position_im, ia SYS.ODCIINDEXINFO,
                                 op SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO,
                                 strt NUMBER, stop NUMBER, lower_pos NUMBER,
                                 upper_pos NUMBER, env SYS.ODCIEnv) RETURN NUMBER,
  MEMBER FUNCTION ODCIINDEXFETCH(SELF IN OUT position_im, nrows NUMBER,
                                 rids OUT SYS.ODCIRIDLIST, env SYS.ODCIEnv)
                                 RETURN NUMBER,
  MEMBER FUNCTION ODCIINDEXCLOSE(env SYS.ODCIEnv) RETURN NUMBER
);
/

CREATE OR REPLACE TYPE BODY position_im
IS
   STATIC FUNCTION ODCIGETINTERFACES(ifclist OUT SYS.ODCIOBJECTLIST)
       RETURN NUMBER IS
   BEGIN
       ifclist := SYS.ODCIOBJECTLIST(SYS.ODCIOBJECT('SYS','ODCIINDEX2'));
       RETURN ODCICONST.SUCCESS;
   END ODCIGETINTERFACES;

 STATIC FUNCTION ODCIINDEXCREATE (ia SYS.ODCIINDEXINFO, parms VARCHAR2, env SYS.ODCIEnv) RETURN
NUMBER
  IS
   stmt   VARCHAR2(2000);
  BEGIN
   -- construct the sql statement
   stmt := 'Create Table ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
           '_STORAGE_TAB' || '(col_val, base_rowid, constraint pk PRIMARY KEY ' ||
           '(col_val, base_rowid)) ORGANIZATION INDEX AS SELECT ' ||
           ia.INDEXCOLS(1).COLNAME || ', ROWID FROM ' ||
           ia.INDEXCOLS(1).TABLESCHEMA || '.' || ia.INDEXCOLS(1).TABLENAME;

   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;
```

```
  STATIC FUNCTION ODCIINDEXDROP(ia SYS.ODCIINDEXINFO, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
    -- construct the sql statement
   stmt := 'DROP TABLE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
    '_STORAGE_TAB';

   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;

  STATIC FUNCTION ODCIINDEXTRUNCATE(ia SYS.ODCIINDEXINFO, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
   -- construct the sql statement
   stmt := 'TRUNCATE TABLE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME || '_STORAGE_TAB';

   EXECUTE IMMEDIATE stmt;
   RETURN ODCICONST.SUCCESS;
  END;

  STATIC FUNCTION ODCIINDEXINSERT(ia SYS.ODCIINDEXINFO, rid ROWID,
                          newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
    -- construct the sql statement
   stmt := 'INSERT INTO ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
          '_STORAGE_TAB  VALUES (''' || newval || ''' , ''' || rid || ''' )';

   -- execute the statement
   EXECUTE IMMEDIATE stmt;

   RETURN ODCICONST.SUCCESS;
  END;

  STATIC FUNCTION ODCIINDEXDELETE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                                  env SYS.ODCIEnv)
     RETURN NUMBER IS
   stmt VARCHAR2(2000);
  BEGIN
    -- construct the sql statement
   stmt := 'DELETE FROM ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
          '_STORAGE_TAB WHERE col_val = ''' || oldval || ''' AND base_rowid = ''' || rid ||
''''';
```

```
 -- execute the statement
 EXECUTE IMMEDIATE stmt;

 RETURN ODCICONST.SUCCESS;
END;

STATIC FUNCTION ODCIINDEXUPDATE(ia SYS.ODCIINDEXINFO, rid ROWID, oldval NUMBER,
                        newval NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS
 stmt VARCHAR2(2000);
BEGIN
  -- construct the sql statement
 stmt := 'UPDATE ' || ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
        '_STORAGE_TAB SET col_val = ''' || newval || ''' WHERE f2 = '''|| rid ||'''';

 -- execute the statement
 EXECUTE IMMEDIATE stmt;

 RETURN ODCICONST.SUCCESS;
END;

STATIC FUNCTION ODCIINDEXSTART(SCTX IN OUT position_im, ia SYS.ODCIINDEXINFO,
                       op SYS.ODCIPREDINFO, qi SYS.ODCIQUERYINFO,
                       strt NUMBER, stop NUMBER, lower_pos NUMBER,
                       upper_pos NUMBER, env SYS.ODCIEnv) RETURN NUMBER IS

  rid              VARCHAR2(5072);
  storage_tab_name VARCHAR2(65);
  lower_bound_stmt VARCHAR2(2000);
  upper_bound_stmt VARCHAR2(2000);
  range_query_stmt VARCHAR2(2000);
  lower_bound      NUMBER;
  upper_bound      NUMBER;
  cnum             INTEGER;
  nrows            INTEGER;

BEGIN
  -- Take care of some error cases.
  -- The only predicates in which position operator can appear are
  --    op() = 1     OR
  --    op() = 0     OR
  --    op() between 0 and 1
  IF (((strt != 1) AND (strt != 0)) OR
      ((stop != 1) AND (stop != 0)) OR
      ((strt = 1) AND (stop = 0))) THEN
    RAISE_APPLICATION_ERROR(-20101,
```

```
                          'incorrect predicate for position_between operator');
END IF;

IF (lower_pos > upper_pos) THEN
  RAISE_APPLICATION_ERROR(-20101, 'Upper Position must be greater than or
  equal to Lower Position');
END IF;

IF (lower_pos <= 0) THEN
  RAISE_APPLICATION_ERROR(-20101, 'Both Positions must be greater than zero');
END IF;

storage_tab_name := ia.INDEXSCHEMA || '.' || ia.INDEXNAME ||
                    '_STORAGE_TAB';
upper_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                    storage_tab_name || ') */ DISTINCT ' ||
                    'col_val FROM ' || storage_tab_name || ' ORDER BY ' ||
                    'col_val DESC) WHERE rownum <= ' || lower_pos;
EXECUTE IMMEDIATE upper_bound_stmt INTO upper_bound;

IF (lower_pos != upper_pos) THEN
  lower_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                      storage_tab_name || ') */ DISTINCT ' ||
                      'col_val FROM ' || storage_tab_name ||
                      ' WHERE col_val < ' || upper_bound || ' ORDER BY ' ||
                      'col_val DESC) WHERE rownum <= ' ||
                      (upper_pos - lower_pos);
  EXECUTE IMMEDIATE lower_bound_stmt INTO lower_bound;
ELSE
  lower_bound := upper_bound;
END IF;

IF (lower_bound IS NULL) THEN
  lower_bound := upper_bound;
END IF;

range_query_stmt := 'Select base_rowid FROM ' || storage_tab_name ||
                    ' WHERE col_val BETWEEN ' || lower_bound || ' AND ' ||
                    upper_bound;

cnum := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(cnum, range_query_stmt, DBMS_SQL.NATIVE);

-- set context as the cursor number
SCTX := position_im(cnum, 0, 0, 0);
```

```
  -- return success
  RETURN ODCICONST.SUCCESS;
END;

MEMBER FUNCTION ODCIINDEXFETCH(SELF IN OUT position_im, nrows NUMBER,
                               rids OUT SYS.ODCIRIDLIST, env SYS.ODCIEnv)
 RETURN NUMBER IS
  cnum    INTEGER;
  rid_tab DBMS_SQL.Varchar2_table;
  rlist   SYS.ODCIRIDLIST := SYS.ODCIRIDLIST();
  i       INTEGER;
  d       INTEGER;
BEGIN
  cnum := SELF.curnum;

  IF self.howmany = 0 THEN
    dbms_sql.define_array(cnum, 1, rid_tab, nrows, 1);
    d := DBMS_SQL.EXECUTE(cnum);
  END IF;

  d := DBMS_SQL.FETCH_ROWS(cnum);

  IF d = nrows THEN
    rlist.extend(d);
  ELSE
    rlist.extend(d+1);
  END IF;

  DBMS_SQL.COLUMN_VALUE(cnum, 1, rid_tab);

  for i in 1..d loop
    rlist(i) := rid_tab(i+SELF.howmany);
  end loop;

  SELF.howmany := SELF.howmany + d;
  rids := rlist;

  RETURN ODCICONST.SUCCESS;
END;

MEMBER FUNCTION ODCIINDEXCLOSE(env SYS.ODCIEnv) RETURN NUMBER IS
  cnum INTEGER;
BEGIN
  cnum := SELF.curnum;
```

```
    DBMS_SQL.CLOSE_CURSOR(cnum);
    RETURN ODCICONST.SUCCESS;
  END;

END;
/
```

The next step is to create the functional implementation `function_for_position_between` for the operator that will be associated with the indextype. (The PL/SQL blocks are shown in parentheses.)

This function is for use with an index-based function evaluation. Therefore, it takes an index context and scan context as parameters.

See Also:

- *Oracle9i Data Cartridge Developer's Guide* for information on creating index-based functional implementation
- CREATE FUNCTION on page 13-49 and *PL/SQL User's Guide and Reference*

```
CREATE OR REPLACE FUNCTION function_for_position_between
                          (col NUMBER, lower_pos NUMBER, upper_pos NUMBER,
                           indexctx IN SYS.ODCIIndexCtx,
                           scanctx IN OUT position_im,
                           scanflg IN NUMBER)
RETURN NUMBER AS
  rid              ROWID;
  storage_tab_name VARCHAR2(65);
  lower_bound_stmt VARCHAR2(2000);
  upper_bound_stmt VARCHAR2(2000);
  col_val_stmt     VARCHAR2(2000);
  lower_bound      NUMBER;
  upper_bound      NUMBER;
  column_value     NUMBER;
BEGIN
  IF (indexctx.IndexInfo IS NOT NULL) THEN
    storage_tab_name := indexctx.IndexInfo.INDEXSCHEMA || '.' ||
                        indexctx.IndexInfo.INDEXNAME || '_STORAGE_TAB';
    IF (scanctx IS NULL) THEN
      --This is the first call.  Open a cursor for future calls.
      --First, do some error checking
      IF (lower_pos > upper_pos) THEN
        RAISE_APPLICATION_ERROR(-20101,
          'Upper Position must be greater than or equal to Lower Position');
```

```
    END IF;
    IF (lower_pos <= 0) THEN
      RAISE_APPLICATION_ERROR(-20101,
        'Both Positions must be greater than zero');
    END IF;
    --Obtain the upper and lower value bounds for the range we're interested
    --in.
    upper_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                        storage_tab_name || ') */ DISTINCT ' ||
                        'col_val FROM ' || storage_tab_name || ' ORDER BY ' ||
                        'col_val DESC) WHERE rownum <= ' || lower_pos;
    EXECUTE IMMEDIATE upper_bound_stmt INTO upper_bound;
    IF (lower_pos != upper_pos) THEN
      lower_bound_stmt := 'Select MIN(col_val) FROM (Select /*+ INDEX_DESC(' ||
                          storage_tab_name || ') */ DISTINCT ' ||
                          'col_val FROM ' || storage_tab_name ||
                          ' WHERE col_val < ' || upper_bound || ' ORDER BY ' ||
                          'col_val DESC) WHERE rownum <= ' ||
                          (upper_pos - lower_pos);
      EXECUTE IMMEDIATE lower_bound_stmt INTO lower_bound;
    ELSE
      lower_bound := upper_bound;
    END IF;
    IF (lower_bound IS NULL) THEN
      lower_bound := upper_bound;
    END IF;
    --Store the lower and upper bounds for future function invocations for
    --the positions.
    scanctx := position_im(0, 0, lower_bound, upper_bound);
  END IF;
  --Fetch the column value corresponding to the rowid, and see if it falls
  --within the determined range.
  col_val_stmt := 'Select col_val FROM ' || storage_tab_name ||
                  ' WHERE base_rowid = ''' || indexctx.Rid || '''';
  EXECUTE IMMEDIATE col_val_stmt INTO column_value;
  IF (column_value <= scanctx.upper_bound AND
      column_value >= scanctx.lower_bound AND
      scanflg = ODCICONST.RegularCall) THEN
    RETURN 1;
  ELSE
    RETURN 0;
  END IF;
ELSE
  RAISE_APPLICATION_ERROR(-20101, 'A column that has a domain index of' ||
                          'Position indextype must be the first argument');
```

```
  END IF;
END;
/
```

Next, create the `position_between` operator, which uses the `function_for_position_between` function. The operator takes an indexed `NUMBER` column as the first argument, followed by a `NUMBER` lower and upper bound as the second and third arguments.

See Also: CREATE OPERATOR on page 14-42

```
CREATE OR REPLACE OPERATOR position_between
  BINDING (NUMBER, NUMBER, NUMBER) RETURN NUMBER
  WITH INDEX CONTEXT, SCAN CONTEXT position_im
  USING function_for_position_between;
```

In this `CREATE OPERATOR` statement, the `WITH INDEX CONTEXT, SCAN CONTEXT position_im` clause is included so that the index context and scan context are passed in to the functional evaluation, which is index based.

Now create the `position_indextype` indextype for the `position_operator`:

See Also: CREATE INDEXTYPE on page 13-91

```
CREATE INDEXTYPE position_indextype
  FOR position_between(NUMBER, NUMBER, NUMBER)
  USING position_im;
```

The operator `position_between` uses an index-based functional implementation. Therefore, a domain index must be defined on the referenced column so that the index information can be passed into the functional evaluation. So the final step is to create the domain index `salary_index` using the `position_indextype` indextype:

See Also: CREATE INDEX on page 13-62

```
CREATE INDEX salary_index ON employees(salary)
   INDEXTYPE IS position_indextype;
```

Now you can use the `position_between` operator function to rewrite the original query as follows:

```
SELECT last_name, salary FROM employees
   WHERE position_between(salary, 10, 20)=1
```

```
   ORDER BY salary DESC;

LAST_NAME                      SALARY
------------------------- ----------
Tucker                          10000
King                            10000
Baer                            10000
Bloom                           10000
Fox                              9600
Bernstein                        9500
Sully                            9500
Greene                           9500
Hunold                           9000
Faviet                           9000
McEwen                           9000
Hall                             9000
Hutton                           8800
Taylor                           8600
Livingston                       8400
Gietz                            8300
Chen                             8200
Fripp                            8200
Weiss                            8000
Olsen                            8000
Smith                            8000
Kaufling                         7900
```

Using XML in SQL Statements

This section describes some of the ways you can use `XMLType` data in the database.

XMLType Tables

The sample schema `oe` contains a table `warehouses`, which contains an `XMLType` column `warehouse_spec`. Suppose you want to create a separate table with the `warehouse_spec` information. The following example creates a very simple `XMLType` table with one implicit `CLOB` column:

```
CREATE TABLE xwarehouses OF XMLTYPE;
```

You can insert into such a table using `XMLType` syntax, as shown in the next statement. (The data inserted in this example corresponds to the data in the `warehouse_spec` column of the sample table `oe.warehouses` where `warehouse_id` = 1.)

```
INSERT INTO xwarehouses VALUES
  (xmltype('<?xml version="1.0"?>
  <Warehouse>
    <WarehouseId>1</WarehouseId>
    <WarehouseName>Southlake, Texas</WarehouseName>
    <Building>Owned</Building>
    <Area>25000</Area>
    <Docks>2</Docks>
    <DockType>Rear load</DockType>
    <WaterAccess>true</WaterAccess>
    <RailAccess>N</RailAccess>
    <Parking>Street</Parking>
    <VClearance>10</VClearance>
  </Warehouse>'));
```

> **See Also:** *Oracle9i XML Database Developer's Guide - Oracle XML DB* for information on `XMLType` and its member methods

You can query this table with the following statement:

```
SELECT e.getClobVal() FROM xwarehouses e;
```

Because Oracle implicitly stores the data in a `CLOB` column, it is subject to all of the restrictions on LOB columns. To avoid these restrictions, create an XMLSchema-based table. The XMLSchema maps the XML elements to their object-relational equivalents. The following example registers an XMLSchema locally. The XMLSchema (`xwarhouses.xsd`) reflects the same structure as the `xwarehouses` table. (XMLSchema declarations use PL/SQL and the `DBMS_XMLSCHEMA` package, so the example is shown in italics.)

> **See Also:** *Oracle9i XML Database Developer's Guide - Oracle XML DB* for information on creating XMLSchemas

```
begin
 dbms_xmlschema.registerSchema(
  'http://www.oracle.com/xwarehouses.xsd',
  '<schema xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace="http://www.oracle.com/xwarehouses.xsd"
       xmlns:who="http://www.oracle.com/xwarehouses.xsd"
       version="1.0">

  <simpleType name="RentalType">
   <restriction base="string">
    <enumeration value="Rented"/>
```

```
    <enumeration value="Owned"/>
   </restriction>
  </simpleType>

  <simpleType name="ParkingType">
   <restriction base="string">
    <enumeration value="Street"/>
    <enumeration value="Lot"/>
   </restriction>
  </simpleType>

  <element name = "Warehouse">
    <complexType>
     <sequence>
      <element name = "WarehouseId"   type = "positiveInteger"/>
      <element name = "WarehouseName" type = "string"/>
      <element name = "Building"      type = "who:RentalType"/>
      <element name = "Area"          type = "positiveInteger"/>
      <element name = "Docks"         type = "positiveInteger"/>
      <element name = "DockType"      type = "string"/>
      <element name = "WaterAccess"   type = "boolean"/>
      <element name = "RailAccess"    type = "boolean"/>
      <element name = "Parking"       type = "who:ParkingType"/>
      <element name = "VClearance"    type = "positiveInteger"/>
     </sequence>
    </complexType>
  </element>
</schema>',
   TRUE, TRUE, FALSE, FALSE);
end;
/
```

Now you can create an XMLSchema-based table, as shown in the following example:

```
CREATE TABLE xwarehouses OF XMLTYPE
   XMLSCHEMA "http://www.oracle.com/xwarehouses.xsd"
   ELEMENT "Warehouse";
```

By default, Oracle stores this as an object-relational table. Therefore, you can insert into it as shown in the example that follows. (The data inserted in this example corresponds to the data in the `warehouse_spec` column of the sample table `oe.warehouses` where `warehouse_id` = 1.)

```
INSERT INTO xwarehouses VALUES(
   xmltype.createxml('<?xml version="1.0"?>
```

```
   <who:Warehouse xmlns:who="http://www.oracle.com/xwarehouse.xsd"
   xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
   xsi:schemaLocation="http://www.oracle.com/xwarehouse.xsd
   http://www.oracle.com/xwarehouse.xsd">
      <WarehouseId>1</WarehouseId>
      <WarehouseName>Southlake, Texas</WarehouseName>
      <Building>Owned</Building>
      <Area>25000</Area>
      <Docks>2</Docks>
      <DockType>Rear load</DockType>
      <WaterAccess>true</WaterAccess>
      <RailAccess>false</RailAccess>
      <Parking>Street</Parking>
      <VClearance>10</VClearance>
      </who:Warehouse>'));
...
```

You can define constraints on an XMLSchema-based table. To do so, you use the `XMLDATA` pseudocolumn to refer to the appropriate attribute within the `Warehouse` XML element:

```
ALTER TABLE xwarehouses ADD (PRIMARY KEY(XMLDATA."WarehouseId"));
```

Because the data in `xwarehouses` is stored object relationally, Oracle rewrites queries to this `XMLType` table to go to the underlying storage when possible. Therefore the following queries would use the index created by the primary key constraint in the preceding example:

```
SELECT * FROM xwarehouses x
   WHERE EXISTSNODE(VALUE(x), '/Warehouse[WarehouseId="1"]') = 1;

SELECT * FROM xwarehouses x
   WHERE EXTRACTVALUE(VALUE(x), '/Warehouse/WarehouseId') = 1;
```

You can also explicitly create indexes on XMLSchema-based tables, which greatly enhance the performance of subsequent queries. You can create object-relational views on `XMLType` tables, and you can create `XMLType` views on object-relational tables.

See Also:

- "XMLDATA" on page 2-90 for information on the XMLDATA pseudocolumn
- "Creating an XMLType View: Example" on page 16-53
- "Create an Index on an XMLType Table: Example" on page 13-84

XMLType Columns

The sample table oe.warehouses was created with a warehouse_spec column of type XMLType. No storage was specified, so the XMLType column was implicitly stored as a CLOB. The examples in this section create a shortened form of the oe.warehouses table, using two different types of storage.

The first example creates a table with an XMLType table stored as a CLOB. This table does not require an XMLSchema, so the content structure is not predetermined:

```
CREATE TABLE xwarehouses (
   warehouse_id        NUMBER,
   warehouse_spec      XMLTYPE)
   XMLTYPE warehouse_spec STORE AS CLOB
   (TABLESPACE demo
    STORAGE (INITIAL 6144 NEXT 6144)
    CHUNK 4000
    NOCACHE LOGGING);
```

The following example creates a similar table, but stores the XMLType data in an object-relational XMLType column whose structure is determined by the specified XMLSchema:

```
CREATE TABLE xwarehouses (
   warehouse_id    NUMBER,
   warehouse_spec  XMLTYPE)
   XMLTYPE warehouse_spec STORE AS OBJECT RELATIONAL
      XMLSCHEMA "http://www.oracle.com/xwarehouses.xsd"
      ELEMENT "Warehouse";
```

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y