# Oracle Rdb7™

# SQL Reference Manual
# Volume 3

Release 7.0

Part No. A42812-1

ORACLE®

SQL Reference Manual, Volume 3

Release 7.0

Part No. A42812-1

# Contents

# Send Us Your Comments

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

You can send comments to us in the following ways:

- Electronic mail — `nedc_doc@us.oracle.com`
- FAX — `603-897-3334 Attn: Oracle Rdb Documentation`
- Postal service

```
Oracle Corporation
Oracle Rdb Documentation
One Oracle Drive
Nashua, NH  03062
USA
```

If you like, you can use the following questionnaire to give us feedback. (Edit the online release notes file, extract a copy of this questionnaire, and send it to us.)

Name ______________________________ Title ______________________

Company ___________________________ Department _________________

Mailing Address ______________________ Telephone Number ____________

_______________________________________________________________

Book Title ___________________________ Version Number ____________

- 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 about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available).

# Preface

This manual describes the syntax and semantics of all the statements and language elements for the SQL (structured query language) interface to the Oracle Rdb database software.

## Intended Audience

To get the most out of this manual, you should be familiar with data processing procedures, basic database management concepts and terminology, and the OpenVMS operating system.

## Operating System Information

You can find information about the versions of the operating system and optional software that are compatible with this version of Oracle Rdb in the *Oracle Rdb7 Installation and Configuration Guide*.

For information on the compatibility of other software products with this version of Oracle Rdb, refer to the *Oracle Rdb7 Release Notes*.

Contact your Oracle representative if you have questions about the compatibility of other software products with this version of Oracle Rdb.

## Structure

This manual is divided into three volumes. Volume 1 contains Chapter 1 through Chapter 5, and an index. Volume 2 contains Chapter 6 and an index. Volume 3 contains Chapter 7, the appendixes, and an index.

The index for each volume contains entries for the respective volume only and does not contain index entries from the other volumes in the set.

The following table shows the contents of the chapters and appendixes in Volumes 1, 2, and 3 of the *Oracle Rdb7 SQL Reference Manual*:

|  |  |
|---|---|
| Chapter 1 | Introduces SQL (structured query language) and briefly describes SQL functions. This chapter also describes conformance to the ANSI standard, how to read syntax diagrams, executable and nonexecutable statements, keywords and line terminators, and support for Multivendor Integration Architecture. |
| Chapter 2 | Describes the language and syntax elements common to many SQL statements. |
| Chapter 3 | Describes the syntax for the SQL module language and the SQL module processor command line. |
| Chapter 4 | Describes the syntax of the SQL precompiler command line. |
| Chapter 5 | Describes SQL routines. |
| Chapter 6 and Chapter 7 | Describes in detail the syntax and semantics of the SQL statements. These chapters include descriptions of data definition statements, data manipulation statements, and interactive control commands. |
| Appendix A | Describes the different types of errors encountered in SQL and where they are documented. |
| Appendix B | Describes the SQL Communications Area and the message vector. |
| Appendix C | Describes the SQLSTATE error handling mechanism. |
| Appendix D | Describes the SQL Descriptor Areas and how they are used in dynamic SQL programs. |
| Appendix E | Summarizes the logical names and configuration parameters that SQL recognizes for special purposes. |
| Appendix F | Summarizes the obsolete SQL features of the current Oracle Rdb version. |
| Appendix G | Summarizes the SQL functions that have been added to the Oracle Rdb SQL interface for convergence with Oracle7 SQL. |
| Index | Volume 3 only. |

## Related Manuals

For more information on Oracle Rdb, see the other manuals in this documentation set, especially the following:

- *Oracle Rdb7 Guide to Database Design and Definition*
- *Oracle Rdb7 Guide to Database Performance and Tuning*
- *Oracle Rdb7 Introduction to SQL*
- *Oracle Rdb7 Guide to SQL Programming*

## Conventions

This manual uses icons to identify information that is specific to an operating system or platform. Where material pertains to more than one platform or operating system, combination icons or generic icons are used. For example:

| | |
|---|---|
| Digital UNIX | This icon denotes the beginning of information specific to the Digital UNIX operating system. |
| OpenVMS VAX OpenVMS Alpha | This icon combination denotes the beginning of information specific to both the OpenVMS VAX and OpenVMS Alpha operating systems. |
| ♦ | The diamond symbol denotes the end of a section of information specific to an operating system or platform. |

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

Often in examples the prompts are not shown. Generally, they are shown where it is important to depict an interactive sequence exactly; otherwise, they are omitted.

Discussions in this manual that refer to VMScluster environments apply to both VAXcluster systems that include only VAX nodes and VMScluster systems that include at least one Alpha node, unless indicated otherwise.

The following conventions are also used in this manual:

| | |
|---|---|
| .<br>.<br>. | Vertical ellipsis points in an example mean that information not directly related to the example has been omitted. |
| . . . | Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted. |
| e, f, t | Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page. |
| **boldface text** | Boldface type in text indicates a new term. |
| < > | Angle brackets enclose user-supplied names in syntax diagrams. |

|  |  |
|---|---|
| [ ] | Brackets enclose optional clauses from which you can choose one or none. |
| $ | The dollar sign represents the command language prompt. This symbol indicates that the command language interpreter is ready for input. |
| UPPERCASE<br>lowercase | The Digital UNIX operating system differentiates between lowercase and uppercase characters. Examples, syntax descriptions, function definitions, and literal strings that appear in text must be typed exactly as shown. |

## References to Products

The Oracle Rdb documentation set to which this manual belongs often refers to the following Oracle Corporation products by their abbreviated names:

- In this manual, Oracle Rdb refers to Oracle Rdb for OpenVMS and Oracle Rdb for Digital UNIX software. Version 7.0 of Oracle Rdb software is often referred to as V7.0.
- The SQL interface to Oracle Rdb is referred to as SQL. This interface is the Oracle Rdb implementation of the SQL standard ANSI X3.135-1992, ISO 9075:1992, commonly referred to as the ANSI/ISO SQL standard or SQL92.
- Oracle CDD/Repository software is referred to as the dictionary, the data dictionary, or the repository.
- Oracle ODBC Driver for Rdb software is referred to as the ODBC driver.
- OpenVMS means both the OpenVMS Alpha and OpenVMS VAX operating system.

# Technical Changes and New Features

This section identifies the new and updated portions of this manual since it was last released with V6.0.

The *Oracle Rdb7 Release Notes* describes current limitations and restrictions.

**The major new features and technical changes for V6.1 that are described in this manual are:**

- INTEGER data type for SQL module language allows modifiers

  The SQL module language syntax has been extended to allow specification of precise INTEGER module parameters in the number of bits.

- New command line qualifiers for SQL module language and precompiled SQL

  Table 1 shows the new qualifiers for SQL module language and precompiled SQL and the appropriate platform.

**Table 1 Command Line Qualifiers**

| Qualifier Name | Digital UNIX | OpenVMS Alpha | OpenVMS VAX |
|---|---|---|---|
| **SQL Module Language** | | | |
| [NO]ALIGN_RECORDS | | X | X |
| –[no]align | X | | |
| [NO]LOWERCASE_PROCEDURE_ NAMES | | X | X |
| –[no]lc_proc | X | | |
| [NO]C_PROTOTYPES | | X | X |
| –[no]cproto | X | | |

(continued on next page)

**Table 1 (Cont.) Command Line Qualifiers**

| Qualifier Name | Digital UNIX | OpenVMS Alpha | OpenVMS VAX |
|---|---|---|---|
| **SQL Module Language** | | | |
| [NO]LONG_SQLCODE | | X | X |
| –[no]lsqlcode | X | | |
| [NO]EXTERNAL_GLOBALS | | X | X |
| –[no]extern | X | | |
| USER_DEFAULT | | X | X |
| –user username | X | | |
| PASSWORD_DEFAULT | | X | X |
| –pass password | X | | |
| [NO]PACKAGE_COMPILATION | | X | X |
| ROLLBACK_ON_EXIT | | X | X |
| –fida | X | | |
| –int32 | X | | |
| –int64 | X | | |
| –plan file-spec | X | | |
| **SQL Precompiler** | | | |
| [NO]DECLARE_MESSAGE_ VECTOR | | X | X |
| –s ′–[no]msgvec′ | X | | |
| USER_DEFAULT | | X | X |
| –s ′–user username′ | X | | |
| PASSWORD_DEFAULT | | X | X |
| –s ′–pass password′ | X | | |
| ROLLBACK_ON_EXIT | | X | X |
| [NO]EXTERNAL_GLOBALS | | X | X |
| –s ′–[no]extern′ | X | | |
| –plan file-spec | X | | |

See Chapter 3 and Chapter 4 for more information.

- Asynchronous creation of storage areas

You can specify whether Oracle Rdb creates storage areas serially, creates a specified number at the same time, or creates all areas at the same time.

For information about the SQL syntax, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Authenticating users for remote access

  Oracle Rdb lets you explicitly provide user name and password information in SQL statements that attach to the database. In addition, it lets you pass the information to an SQL module language or precompiled SQL program by using a parameter and new command line qualifiers. You can also pass the information to Oracle Rdb by using configuration parameters.

- Selecting an outline to use for a query

  Using SQL syntax, you can specify the name of an outline to use for a query.

  SQL statements affected by this feature are DECLARE CURSOR, DELETE, INSERT, SELECT, and UPDATE and select expression.

OpenVMS VAX OpenVMS Alpha

- Notification of classes of operators

  Using SQL syntax, you can specify which classes of operators are notified in the case of a catastrophic journaling event such as running out of disk space. (This feature was available in V6.0 using the RMU interface.)

  For information about the SQL syntax, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement. ♦

- Specifying shutdown time

  Using SQL syntax, you can specify the number of minutes the database system will wait after a catastrophic event before it shuts down the database. (This feature was available in V6.0 using the RMU interface.)

  For information about the SQL syntax, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Asynchronous batch-writes

  Using SQL syntax, you can specify that processes write batches of modified data pages to disk asynchronously (the process does not stall while waiting for the batch-write operation to complete). Asynchronous batch-writes improve the performance of update applications without the loss of data integrity. (This feature was available in V6.0 using logical names to specify the number of buffers used.)

For information about the SQL syntax, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Asynchronous prefetch

  Using SQL syntax, you can specify whether or not Oracle Rdb reduces the amount of time that a process waits for pages to be read from disk by fetching pages before a process actually requests the pages.

  For information about the SQL syntax, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Fast incremental backup

  Using SQL syntax, you can specify whether Oracle Rdb checks each area's SPAM pages or each database page to find changes during incremental backup.

  For information about the SQL syntax, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Support for two new character sets

  Oracle Rdb includes support for two new character sets: BIG5 and TACTIS. BIG5 is a fixed 2-octet character set. TACTIS is a single-octet character set.

- TRIM built-in function

  The TRIM built-in function lets you remove leading and trailing characters from a character string.

- POSITION built-in function

  The POSITION built-in function lets you search for a particular substring within another string.

OpenVMS VAX OpenVMS Alpha

- INTEGRATE statement has new arguments

  Oracle Rdb provides a finer level of definition integration between an Oracle Rdb database and the CDD/Repository with the introduction of the DOMAIN and TABLE arguments to the INTEGRATE statement.
  In previous versions of Oracle Rdb, the INTEGRATE statement let you integrate all Oracle Rdb database schema objects with the CDD/Repository but did not allow the integration of individual schema objects. With Oracle Rdb V6.1, the INTEGRATE statement lets you select specific Oracle Rdb schema objects (tables and domains) for integration. However, SQL

continues to let you integrate an entire database with the INTEGRATE statement when that level of integration is required. ♦

- SHOW DATABASE statement includes new information

  The output from the SHOW DATABASE statement includes information about the new database attributes, such as asynchronous batch-writes and shutdown time.

- LIKE predicate optimization in SQL queries

  Oracle Rdb has improved the performance of certain types of LIKE predicates in SQL queries.

- Multistring comments

  You can now specify comments that contain more than one string literal separated by a slash mark (/). This was implemented as a workaround to the limitation that comments can only be 1,024 characters in length. Statements affected by this new feature are:

  – COMMENT ON Statement
  – CREATE COLLATING SEQUENCE Statement
  – CREATE DATABASE Statement
  – CREATE FUNCTION Statement
  – CREATE MODULE Statement
  – CREATE OUTLINE Statement
  – CREATE PROCEDURE Statement

- New UNDECLARE Variable Statement

  You can now undeclare variables. See the UNDECLARE Variable Statement for more information.

- Three logical names introduced in Oracle Rdb V6.0 are deprecated and replaced with new names in V6.1. Table 2 shows the changes

**Table 2 Logical Name Changes**

| V6.0 OpenVMS Logical Name | V6.1 OpenVMS Logical Name | V6.1 Digital UNIX Configuration Parameter |
|---|---|---|
| RDM$BIND_ABW_DISABLED | RDM$BIND_ABW_ENABLED | RDB_BIND_ABW_ENABLED |
| RDM$BIND_APF_DISABLED | RDM$BIND_APF_ENABLED | RDB_BIND_APF_ENABLED |
| RDM$BIND_STATS_DISABLED | RDM$BIND_STATS_ENABLED | RDB_BIND_STATS_ENABLED |

SQL syntax has been introduced in Oracle Rdb V6.1 for these features. Oracle Rdb recommends that you use the SQL syntax for these features. See CREATE DATABASE Statement and ALTER DATABASE Statement for more information regarding the new syntax.

See Appendix E for more information regarding the new logical names.

- Portable SQL routines

  SQL provides the following routines for use on both OpenVMS and Digital UNIX operating systems. For more information, see the Routines topic under help for interactive SQL.

  - sql_close_cursors

    This routine closes all cursors. It functions the same as the SQL$CLOSE_CURSORS routine, which is available only on OpenVMS.

    On Digital UNIX, this routine is case sensitive and must be entered in lowercase.

  - sql_get_error_text

    This routine passes error text with formatted output to programs for processing. It is similar to the SQL$GET_ERROR_TEXT routine, which is available only on OpenVMS systems.

  - sql_get_message_vector

    This routine retrieves information from the message vector about the status of the last SQL statement.

    On Digital UNIX, this routine is case sensitive and must be entered in lowercase.

  - sql_get_error_handler, sql_register_error_handler, and sql_deregister_error_handler

    These routines now work on Digital UNIX, but otherwise have not changed from previous versions of Oracle Rdb.

  - sql_signal

This routine signals that an error has occurred on the execution of an SQL statement. It is equivalent to the SQL$SIGNAL routine, which is available only on OpenVMS systems.

Digital UNIX

- On Digital UNIX, the GLOBAL and EXTERNAL options of the DECLARE ALIAS statement differ.
  - GLOBAL

    Defines the alias to be globally visible
  - EXTERNAL

    Declares an external reference of the alias ♦

**The major new features and technical changes for V7.0 of Oracle Rdb that are described in this manual are:**

- Ranked B-tree structure

  Oracle Rdb now supports a new ranked B-tree structure that allows better optimization of queries, particularly queries involving range retrievals. Oracle Rdb is able to make better estimates of cardinality, reducing disk I/O and lock contention. To create a ranked B-tree structure, use the RANKED keyword of the CREATE INDEX . . . TYPE IS SORTED statement.

  A sorted ranked index allows storage of many records in a small space when you compress duplicates, using the DUPLICATES ARE COMPRESSED clause of the CREATE INDEX statement.

  For additional information, see the CREATE INDEX Statement.

OpenVMS Alpha

- System space global buffers

  Oracle Rdb for OpenVMS Alpha provides a new type of global buffer called system space buffers (SSB). The system space global buffer is located in the OpenVMS Alpha system space, which means that a system space global buffer is fully resident in memory and does not affect the quotas of the working set of the process. As a result, a process referencing a system space global buffer has an additional 256Mb of resident working set space.

  You can specify whether database root global buffers are created in system space or process space by using the SHARED MEMORY clause.

  See the ALTER DATABASE Statement, the CREATE CACHE Clause, the CREATE DATABASE Statement, and the IMPORT Statement for more information. ♦

OpenVMS Alpha ☰

- Specifying if large memory is used to manage the row cache

  The LARGE MEMORY clause specifies if large memory is used to manage the row cache. Large memory allows Oracle Rdb to use as much physical memory as is available and to dynamically map it to the virtual address space of database users. It provides access to a large amount of physical memory through small virtual address windows.

  See the ALTER DATABASE Statement and the CREATE CACHE Clause for more information. ♦

- Row-level memory cache

  The row-level memory cache feature allows frequently referenced rows to remain in memory even when the associated page has been flushed back to disk. This saves in memory usage because only the more recently referenced rows are cached versus caching the entire buffer.

  See the CREATE CACHE Clause, the ALTER DATABASE Statement, the CREATE DATABASE Statement, the CREATE STORAGE AREA Clause, and the IMPORT Statement for more information regarding the row cache areas.

OpenVMS Alpha ☰

- Specifying the number of window panes used by the large memory mapping algorithm

  See the ALTER DATABASE Statement and the CREATE CACHE Clause for more information. ♦

- Specifying if Oracle Rdb replaces rows in the cache when it becomes full

  See the ALTER DATABASE Statement and the CREATE CACHE Clause for more information.

- Specifying the FROM clause in the CREATE OUTLINE statement

  The process for creating outlines has been simplified with the new FROM syntax. You can now specify the statement for which you need an outline within the CREATE OUTLINE statement.

  See the CREATE OUTLINE Statement for more information.

- Freezing data definition changes

  You can ensure that the data definition of your database does not change by using the METADATA CHANGES ARE DISABLED clause of the ALTER DATABASE, CREATE DATABASE, or IMPORT statements.

  See the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement for more information regarding freezing data definition changes.

- Modifying the database buffer size

  You can now modify the database buffer size by using the BUFFER SIZE clause in the ALTER DATABASE statement. In previous versions, you could specify the clause only in the CREATE DATABASE statement.

  See the ALTER DATABASE Statement for more information regarding modifying the database buffer size.

- Creating a default storage area

  You can separate user data from the system data, such as the system tables, by using the DEFAULT STORAGE AREA clause of the CREATE DATABASE or IMPORT statements. This clause specifies that all user data and indexes that are not mapped explicitly to a storage area are stored in the default storage area.

  See the CREATE DATABASE Statement and the IMPORT Statement for more information regarding the default storage area.

- Deleting a storage area with a cascading delete

  You can specify that Oracle Rdb delete a storage area with a cascading delete. When you do, Oracle Rdb deletes database objects referring to the storage area.

  For more information, see the ALTER DATABASE Statement.

- Specifying how a database opens when you create the database

  You can specify whether a database opens automatically or manually when you create the database. In previous versions, you could specify the OPEN IS clause only in the ALTER DATABASE statement.

  See the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement for more information.

- Specifying how long to wait before closing a database

  You can specify how long Oracle Rdb waits before closing the database, by using the WAIT n MINUTES FOR CLOSE clause.

  See the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement for more information.

- Extending the allocation of storage areas

  You can now manually force the storage area to extend by using the ALLOCATION IS clause of the alter-storage-area-params clause.

  See the ALTER DATABASE Statement for more information.

- Vertical partitioning

  You can now partition a table vertically as well as horizontally. When you partition a table horizontally, you divide the rows of the table among storage areas according to data values in one or more columns. A given storage area then contains only those rows whose column values fall within the range that you specify. When you partition a table vertically, you divide the columns of the table among storage areas. A given storage area then contains only some of the columns of a table. Consider using vertical partitioning when you know that access to some of the columns in a table is frequent, but that the access to other columns is occasional.

  For more information, see the CREATE STORAGE MAP Statement.

- Strict partitioning

  You can now specify whether a partitioning key for a storage map is updatable or not updatable. If you specify that the key is not updatable, Oracle Rdb retrieval performance improves because Oracle Rdb can use the partitioning criteria when optimizing the query.

  For more information, see the CREATE STORAGE MAP Statement.

- Quickly deleting data in tables

  If you want to quickly delete the data in a table, but you want to maintain the metadata definition of the table (perhaps to reload the data into a new partitioning scheme), you can use the TRUNCATE TABLE statement.

  For more information, see the TRUNCATE TABLE Statement.

- Creating temporary tables

  You can create temporary tables to store temporary results only for a short duration, perhaps to temporarily store the results of a query so that your application can act on the results of that query. The data in a temporary table is deleted at the end of an SQL session.

  For more information, see the CREATE MODULE Statement, the CREATE TABLE Statement, and the DECLARE LOCAL TEMPORARY TABLE Statement.

- Removing the links with the repository

  You can remove the link between the repository and database but still maintain the data definitions in both places, using the DICTIONARY IS NOT USED clause of the ALTER DATABASE statement.

  For more information, see the ALTER DATABASE Statement.

- Specifying the location of the recovery journal file

  You can specify the location of the recovery journal using the RECOVERY JOURNAL (LOCATION IS 'directory-spec') clause when you alter, create, or import a database.

  For more information, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Specifying an edit string in an .aij backup file name

  You can specify if the backup file name includes an edit string with the EDIT STRING clause of the ALTER DATABASE statement.

  For more information, see the ALTER DATABASE Statement.

- Increasing the fanout factor for adjustable lock granularity

  Adjustable lock granularity for previous versions of Oracle Rdb defaulted to a count of 3. This means that the lock fanout factor was (10, 100, 1000). As databases grow larger, it is becoming necessary to allow these fanout factors to grow to reduce lock requirements for long queries. You can now change the fanout factor by specifying the COUNT IS clause with the ADJUSTABLE LOCK GRANULARITY IS ENABLED clause.

  For more information, the see ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Collecting a workload profile

  A workload profile is a description of the interesting table and column references used by queries in a database work load. When workload collection is enabled, the optimizer collects and records these references in the RDB$WORKLOAD system table.

  For more information, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Collecting cardinality updates

  When cardinality collection is enabled, the optimizer collects cardinalities for the table and non-unique indexes as rows are inserted or deleted from tables. The cardinalities are stored in the RDB$CARDINALITY column of the RDB$RELATIONS, RDB$INDICES, and RDB$INDEX_SEGMENTS system tables. Cardinality collection is enabled by default.

  For more information, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Specifying detected asynchronous prefetch with a threshold value

  Detected asynchronous prefetch can significantly improve performance by using heuristics to determine if an I/O pattern is sequential in behavior even if not actually performing sequential I/O. For example, when fetching a LIST OF BYTE VARYING column, the heuristics detect that the pages being fetched are sequential and fetch ahead asynchronously to avoid wait times when the page is really needed.

  For more information, see the ALTER DATABASE Statement, the CREATE DATABASE Statement, and the IMPORT Statement.

- Setting debug flags using SQL

  A new SET FLAGS statement has been added to interactive and dynamic SQL, and a SHOW FLAGS statement to interactive SQL. The new SET FLAGS statements has been added to enable and disable the database systems debug flags during execution. For more information, see the SET FLAGS Statement and the SHOW Statement.

- Cursors can now stay open across transactions (holdable cursors)

  SQL cursors can now remain open across transaction boundaries. The WITH HOLD clause of the DECLARE CURSOR statement indicates that the cursor will remain open after the transaction ends. A holdable cursor that has been held open retains its position when a new SQL transaction is begun.

  You can also specify the attributes of the holdable cursor as a database default using the SET HOLD CURSORS statement.

  For more information, see the DECLARE CURSOR Statement and the SET HOLD CURSORS Statement.

- External routine enhancements

  Starting with V7.0, external routines can now contain SQL statements to bind to new schema instances and perform database operations. External routine activation, execution, and exception handling is controlled by a new executor manager process.

  External routines are external functions or external procedures that are written in a 3GL language such as C or FORTRAN, linked into a shareable image, and registered in a database schema. External procedures are new in V7.0.

  External routines are available on all platforms.

  For more information, see Section 2.6.4 and the Create Routine Statement.

- Creating stored functions

  In addition to stored procedures, you can now define stored functions using the CREATE MODULE statement. A stored function is invoked by using the function name in a value expression.

  For more information, see the CREATE MODULE Statement, the Compound Statement, and the RETURN Control Statement.

- Returning the value of a stored function

  SQL provides the RETURN statement, which returns the result of a stored function.

  See the RETURN Control Statement for more information.

- DROP MODULE CASCADE and DROP MODULE RESTRICT implemented

  See the DROP MODULE Statement for more information.

- DROP PROCEDURE and DROP FUNCTION for external routines and stored routines implemented

  See the Drop Routine Statement for more information.

- CALL statement in a compound statement

  You can now use the CALL statement within a compound statement and, therefore, in a stored procedure or function to call another stored procedure.

  The CALL statement can also invoke external procedures.

  For more information, see the CALL Statement for Compound Statements.

- New SIGNAL statement

  SQL now adds a new SIGNAL statement for use within a compound statement.

  SIGNAL accepts a single character value expression that is used as the SQLSTATE. The current routine and all calling routines are terminated and the signaled SQLSTATE is passed to the application.

  For more information, see the SIGNAL Control Statement.

- Using the DEFAULT clause, CONSTANT clause, and UPDATABLE clause when declaring variables within compound statements

  Oracle Rdb includes full support in SQL for the CONSTANT, UPDATABLE, and DEFAULT clauses on declared variables within compound statements.

The default can be any value expression including subqueries, conditional, character, date/time, and numeric expressions. Additionally, Oracle Rdb can now inherit the default from the named domain if one exists.

The CONSTANT clause changes the variable into a declared constant that cannot be updated. If you use the CONSTANT clause, you must also have used the DEFAULT clause to ensure the variable has a value.

The UPDATABLE clause allows a variable to be updated through a SET assignment, an INTO assignment (as part of an INSERT, UPDATE, or SELECT statement), an equality (=) comparison, or as a parameter to a procedure OUT or INOUT parameter.

For more information, see the Compound Statement.

- Obtaining the connection name using the GET DIAGNOSTIC statement

  You can now obtain the current connection name in a variable or parameter from within a stored function, stored procedure, and a multistatement block using the GET DIAGNOSTICS statement.

  For more information, see the GET DIAGNOSTICS Statement.

- Support for the Shift_JIS character set

  Oracle Rdb includes support for the Shift_JIS character set; a mixed multi-octet character set.

  See Section 2.1 for more information.

- Altering RDB$SYSTEM storage area

  You can specify RDB$SYSTEM as the storage area name in the ALTER STORAGE AREA clause of an ALTER DATABASE statement. See ALTER DATABASE Statement for more information.

- Enhancements for the SQL SHOW statement

  The SQL SHOW statement displays the new features affecting data definition, stored routines, and external routines.

  For more information, see the SHOW Statement.

- The keyword ROWID

  You can use keyword ROWID as a synonym for the keyword DBKEY.

- COUNT function enhancements

  You can now specify:

  - COUNT (*)
  - COUNT (value-expr)

  - COUNT (DISTINCT value-expr)

  See Section 2.6.3.1 for more detail.

- Specifying the new dialect ORACLE LEVEL1

  You can now specify the ORACLE LEVEL1 dialect for the interactive SQL and dynamic SQL environments. This dialect is similar to the SQL92 dialect. For more information, see SET DIALECT Statement.

- Two new basic predicates added for inequality comparisons

  These new basic predicates are:

  ```
  ^=
  !=
  ```

  The != predicate is available only if you set your dialect to ORACLE LEVEL1. See Section 2.7.1 for more information on basic predicates.

- Enhancements to the NULL keyword

  The NULL keyword can be used as a value expression. For example, in a SELECT statement. See Section 2.6.1.

OpenVMS VAX OpenVMS Alpha

- Specifying C_PROTOTYPES=file-name

  The SQL module language C_PROTOTYPES qualifier now accepts a file name. See Section 3.5 for more information. ♦

Digital UNIX

- Editing in interactive SQL

  On Digital UNIX, you can use the EDIT statement within interactive SQL. It works similar to the SQL EDIT statement on OpenVMS. For more information, see the EDIT Statement. ♦

Digital UNIX

- Support for Pascal and FORTRAN on Oracle Rdb for Digital UNIX

  Oracle Rdb for Digital UNIX now supports the DEC FORTRAN and DEC Pascal languages for the SQL precompiler and the SQL module processor. ♦

- New command line qualifier for precompiled SQL

  Precompiled SQL now has the –[no]extend_source qualifier on the Digital UNIX platform.

  See Chapter 4 for more information.

# 7 SQL Statements

This chapter describes the syntax and semantics of all statements in SQL. SQL statements include data definition statements; data manipulation statements; statements that control the environment and program flow; and statements that give information.

See Chapter 2 in Volume 1 for detailed descriptions of the language and syntax elements referred to by the syntax diagrams in this chapter.

## DELETE Statement

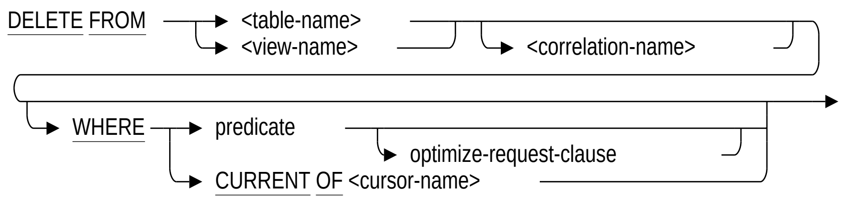
Deletes a row from a table or view.

### Environment

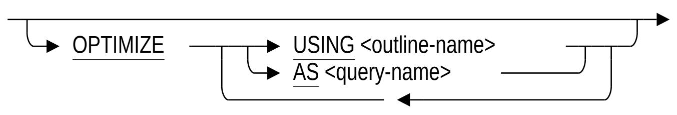
You can use the DELETE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



optimize-request-clause =



### Arguments

**table-name**
**view-name**
Specifies the name of the target table or view from which you want to delete a row.

**correlation name**
Specifies a name that identifies the table or view in the predicate of the DELETE statement. See Section 2.2.10.1 for more information about correlation names.

**WHERE**
Specifies the rows of the target table or view that will be deleted. If you omit the WHERE clause, SQL deletes all rows of the target table or view. You can specify either a predicate or a cursor name in the WHERE clause.

**predicate**
If the WHERE clause includes a predicate, all the rows of the target table for which the predicate is true are deleted. See Section 2.7 for more information on predicates.

**OPTIMIZE USING outline-name**
Names the query outline to be used with the DELETE statement even if the outline ID for the query and for the outline are different.

A **query outline** is an overall plan for how a query can be implemented. See the CREATE OUTLINE Statement for additional information.

**OPTIMIZE AS query-name**
Assigns a name to the query.

**CURRENT OF cursor-name**
If the WHERE clause uses CURRENT OF cursor-name, SQL deletes only the row on which the named cursor is positioned.

The cursor must have been named previously in a DECLARE CURSOR statement, must be open, and must be positioned on a row. In addition, the FROM clause of the SELECT statement within the DECLARE CURSOR statement must refer to the table or view that is the target of the DELETE statement.

## Usage Notes

- When specifying a column name, if the column name is the same as a parameter, you must use a correlation name or table name with the column name.
- The CURRENT OF clause in an embedded DELETE statement cannot name a cursor based on a dynamic SELECT statement. To refer to a cursor based on a dynamic SELECT statement in the CURRENT OF clause, prepare and dynamically execute the DELETE statement as well.
- The CURRENT OF clause in an embedded DELETE statement cannot name a read-only cursor. See the Usage Notes in the DECLARE CURSOR Statement for information about which cursors are read-only.

- You cannot specify the OPTIMIZE USING or the OPTIMIZE AS clause with the WHERE CURRENT OF clause.
- You cannot specify an outline name in a compound-use-statement. See the Compound Statement for more information about compound statements.
- If an outline exists, Oracle Rdb will use the outline specified in the OPTIMIZE USING clause unless one or more of the directives in the outline cannot be followed. SQL issues an error message if the existing outline cannot be used.

  If you specify the name of an outline that does not exist, Oracle Rdb compiles the query, ignores the outline name, and searches for an existing outline with the same outline ID as the query. If an outline with the same outline ID is found, Oracle Rdb attempts to execute the query using the directives in that outline. If an outline with the same outline ID is not found, the optimizer selects a strategy for the query for execution.

  See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information regarding query outlines.

## Examples

Example 1: Deleting all information about an employee

To delete all the information about an employee, you need to delete rows from several tables within a single transaction. This program fragment deletes the rows from all the result tables that contain information about an employee. Note that all the DELETE operations are included in one transaction so that no employee's records are only partially deleted.

```
       DISPLAY "Enter the ID number of employee".
       DISPLAY "whose records you want to delete:  "
          WITH NO ADVANCING.
       ACCEPT EMP-ID.

EXEC SQL
        DECLARE TRANSACTION READ WRITE
        RESERVING EMPLOYEES      FOR PROTECTED WRITE,
                  JOB_HISTORY    FOR PROTECTED WRITE,
                  SALARY_HISTORY FOR PROTECTED WRITE,
                  DEGREES        FOR PROTECTED WRITE
END-EXEC

EXEC SQL
        DELETE FROM EMPLOYEES E
        WHERE E.EMPLOYEE_ID = :EMP-ID
END-EXEC
```

```
IF SQLCODE < 0 THEN
        EXEC SQL        ROLLBACK        END-EXEC
        GO TO ERROR-PAR
END-IF

EXEC SQL
        DELETE FROM JOB_HISTORY JH
        WHERE JH.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
        EXEC SQL        ROLLBACK        END-EXEC
        GO TO ERROR-PAR
END-IF

EXEC SQL
        DELETE FROM SALARY_HISTORY SH
        WHERE SH.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
        EXEC SQL        ROLLBACK        END-EXEC
        GO TO ERROR-PAR
END-IF

EXEC SQL
        DELETE FROM DEGREES D
        WHERE D.EMPLOYEE_ID = :EMP-ID
END-EXEC

IF SQLCODE < 0 THEN
        EXEC SQL        ROLLBACK        END-EXEC
        GO TO ERROR-PAR
END-IF
```

Example 2: Deleting selected rows from a table

The following statement deletes all rows from the EMPLOYEES table where the employee SALARY_AMOUNT is greater than $75,000. The EMPLOYEES and SALARY_HISTORY tables are both in the database with the alias PERS.

```
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> DELETE FROM PERS.EMPLOYEES E
cont> WHERE EXISTS ( SELECT *
cont>                FROM   PERS.SALARY_HISTORY S
cont>                WHERE  S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                AND    S.SALARY_AMOUNT > 75000
cont>              ) ;
7 rows deleted
```

Example 3: Deleting rows from a table specifying an outline name

The following example shows the syntax used to define the DEL_EMP_75000 outline:

```
SQL> CREATE OUTLINE DEL_EMP_75000
cont>  FROM
cont>     (DELETE FROM EMPLOYEES E
cont>      WHERE EXISTS ( SELECT *
cont>                FROM   SALARY_HISTORY S
cont>                WHERE  S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                AND    S.SALARY_AMOUNT > 75000
cont>              );
```

The following query specifies the DEL_EMP_75000 outline:

```
SQL> DELETE FROM EMPLOYEES E
cont> WHERE EXISTS ( SELECT *
cont>                FROM   SALARY_HISTORY S
cont>                WHERE  S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                AND    S.SALARY_AMOUNT > 75000
cont>              )
cont> OPTIMIZE USING DEL_EMP_75000;
~S: Outline DEL_EMP_75000 used
   .
   .
   .
7 rows deleted
```

## DESCRIBE Statement

Writes information about a prepared statement to the SQL Descriptor Area (SQLDA).

The DESCRIBE statement is a dynamic SQL statement. **Dynamic SQL** lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not part of a program's source code but are generated while the program runs. Dynamic SQL is useful when you cannot predict the type of SQL statement your program needs to process.

The **SQLDA** is a collection of host language variables used only in dynamic SQL programs. To use the SQLDA, host languages must support pointer variables that provide indirect access to storage by storing the address of data instead of directly storing data in the host language variable. The languages supported by the SQL precompiler that also support pointer variables are Ada, C, and PL/I. Any other language that supports pointer variables can use the SQLDA, but must call SQL module procedures that contain SQL statements instead of embedding the SQL statements directly in source code. The SQLDA provides information about dynamic SQL statements to the program and information about memory allocated by the program to SQL.

The DESCRIBE statement is how SQL writes information that the program uses to the SQLDA. Specifically, the DESCRIBE statement stores in the SQLDA the number and data types of any select list items or parameter markers in a prepared statement.

Appendix D describes in more detail the specific fields of the SQLDA and how programs use it to communicate about select list items and parameter markers in prepared statements.

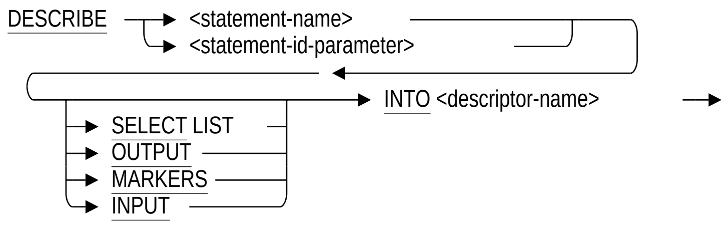
### Environment

You can use the DESCRIBE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

## DESCRIBE Statement

### Format



### Arguments

**statement-name**
**statement-id-parameter**
Specifies the name of a prepared statement. If the PREPARE statement for the dynamically executed statement specifies a parameter, use the same parameter in the DESCRIBE statement instead of an explicit statement name.

You can supply either a parameter or a compile-time statement name. Specifying a parameter lets SQL supply identifiers to programs at run time. Use an integer parameter to contain the statement identifier returned by SQL or a character string parameter to contain the name of the statement that you pass to SQL. See the PREPARE Statement and the DECLARE CURSOR Statement, Dynamic for more details.

**INTO descriptor-name**
Specifies the name of a structure declared in the host language program as an SQLDA to which SQL writes information about select list items, or input or output parameter markers.

Precompiled programs can use the embedded SQL statement INCLUDE SQLDA to automatically insert a declaration of an SQLDA structure, called SQLDA, in the program when it precompiles the program. Programs that use the SQL module language must explicitly declare an SQLDA. Either precompiled or SQL module language programs can explicitly declare additional SQLDAs but must declare them with unique names. For sample declarations of SQLDA structures, see Section D.3.

**SELECT LIST**
**OUTPUT**
Specifies that the DESCRIBE statement writes information about returned values in a prepared statement to the SQLDA. If you use this clause, the DESCRIBE statement writes information about the number and data types of any returned values in the prepared statement to the SQLDA. The program

uses that information to allocate storage for the returned values. The storage allocated by the program then receives the returned values.

The following statements or clauses return values to the DESCRIBE statement:

- Select list items in a SELECT statement
- The following statements within multistatement procedures:
  - Singleton SELECT statement
  - INSERT . . . RETURNING and UPDATE . . . RETURNING statements
  - SET assignment statement
- CALL statement (invoking a stored procedure)
- Dynamic singleton SELECT statement

The default is SELECT LIST (or OUTPUT).

**MARKERS**
**INPUT**
Specifies that the DESCRIBE statement writes information about input parameter markers to the SQLDA. The MARKERS or INPUT clause specifies that the DESCRIBE statement writes information about the number and data types of any input parameter markers in the prepared statement to the SQLDA.

Input parameter markers in a prepared statement serve the same purpose as host language variables in nondynamic, embedded SQL statements. The program can use that information in the SQLDA to allocate storage. The program must supply values in that allocated storage. SQL substitutes these values for the parameter markers when it dynamically executes the prepared statement.

## Usage Notes

- Programs can set values for any fields in the SQLDA in addition to or instead of having SQL set the values in a DESCRIBE statement. SQL uses the values set by the program.

Digital UNIX

- Only the C, COBOL, FORTRAN, and Pascal languages are supported on Digital UNIX. ♦

## Examples

OpenVMS VAX OpenVMS Alpha

Example 1: Using the DESCRIBE . . . SELECT LIST statement with a prepared SELECT statement

This PL/I program illustrates using the DESCRIBE . . . SELECT LIST statement to write information to the SQLDA about the select list items of a prepared SELECT statement. There are no parameter markers in this particular prepared SELECT statement.

After issuing the DESCRIBE statement, the program stores in the SQLDA the addresses of host language variables that will receive values from columns of the result table during FETCH statements.

To shorten the example, this PL/I program is simplified:

- The program includes the SELECT statement to be dynamically executed as part of the source code directly in the PREPARE statement. A program with such coded SQL statements does not need to use dynamic SQL at all, but can simply embed the SELECT statement in a DECLARE CURSOR statement. (A program that must process SQL statements generated as it executes is the only type that requires dynamic SQL.)
- The program declares host language variables for select list items without checking the SQLDA for a description of those items. Typically, an application needs to look in the SQLDA to determine the number and data type of select list items generated by a prepared SELECT statement before allocating storage.
- The program does not use the DESCRIBE . . . MARKERS statement to determine if there are any parameter markers in this dynamically executed SELECT statement. In this example, because the SELECT statement is coded in the program, it is clear that there is no need for a DESCRIBE . . . MARKERS statement. However, if the SELECT statement is generated at run time, the program may have to determine if there are parameter markers by issuing a DESCRIBE . . . MARKERS statement and looking at the value of the SQLD field in the SQLDA.

```
CURSOR_EX : PROCEDURE OPTIONS (MAIN);
/*
* Illustrate the DESCRIBE...SELECT LIST statement using a
* dynamic SELECT statement:
*
* Use a dynamic SELECT statement as the basis for
* a cursor that displays a join of the EMPLOYEES
* and SALARY_HISTORY tables on the screen.
*/
declare sys$putmsg external entry
(any, any value, any value, any value);

/* Declare SQL Communications Area: */
EXEC SQL INCLUDE SQLCA;

/* Declare SQL Descriptor Area: */
EXEC SQL INCLUDE SQLDA;

/* Declare the alias: */
EXEC SQL DECLARE ALIAS FILENAME 'SQL$DATABASE';

/*
 * Branch to ERR_HANDLER if the SQLCODE field
 * of the SQLCA is greater than 0:
 */
EXEC SQL WHENEVER SQLERROR GOTO ERR_HANDLER;

/*
 * Declare a cursor named EMP that uses the
 * prepared statement DYN_SELECT:
 */
EXEC SQL DECLARE EMP CURSOR FOR DYN_SELECT;

/* Declare a host structure to receive
 * the results of FETCH statements:
 */
DCL 1 P_REC,
        2 EMPLOYEE_ID   CHAR(5),
        2 FIRST_NAME    CHAR(10),
        2 LAST_NAME     CHAR(14),
        2 SALARY_AMOUNT FIXED BINARY(31);

/* Allocate memory for the SQLDA and
 * set the value of its SQLN field:
 */
SQLSIZE = 10;
ALLOCATE SQLDA SET (SQLDAPTR);
SQLN = 10;
```

```
/* Prepare the SELECT statement
 * for dynamic execution directly
 * from a statement string:
 */
EXEC SQL PREPARE DYN_SELECT FROM
        'SELECT E.EMPLOYEE_ID,
                E.FIRST_NAME,
                E.LAST_NAME,
                S.SALARY_AMOUNT
        FROM  EMPLOYEES E, SALARY_HISTORY S
        WHERE E.EMPLOYEE_ID = S.EMPLOYEE_ID AND
              S.SALARY_END IS NULL';

/* Write information about the
 * columns of the result table
 * of DYN_SELECT into the SQLDA:
 */
EXEC SQL DESCRIBE DYN_SELECT SELECT LIST INTO SQLDA;

/*
 * Assign the addresses of the host language
 * variables that will receive the values of the
 * fetched row to the SQLDATA field
 * of the SQLDA:
 */
SQLDATA(1) = ADDR( EMPLOYEE_ID );
SQLDATA(2) = ADDR( FIRST_NAME);
SQLDATA(3) = ADDR( LAST_NAME );
SQLDATA(4) = ADDR( SALARY_AMOUNT);

/* Open the cursor: */
EXEC SQL OPEN EMP;

/* Fetch the first row of the result table.
 * SQL uses the addresses in the SQLDA
 * to store values from the table into
 * the host language variables.
 */
EXEC SQL FETCH EMP USING DESCRIPTOR SQLDA;

PUT EDIT ('Current Salaries of Employees: ') (SKIP, A, SKIP(2));

/* While the SQLCODE field of the
 * SQLCA is not 100 (NOT_FOUND error):
 */
DO WHILE (SQLCA.SQLCODE = 0);

        /* Display the values from the host language variables: */
        PUT SKIP EDIT
        (EMPLOYEE_ID, ' ', FIRST_NAME, ' ', LAST_NAME, ' ',
         SALARY_AMOUNT)
        (A, A, A, A, A, A, F(9));
        /* Fetch another row of the result table: */
        EXEC SQL FETCH EMP USING DESCRIPTOR SQLDA;
END;

/* Close the cursor: */
EXEC SQL CLOSE EMP;

RETURN;

ERR_HANDLER:
        PUT EDIT
        ('Unexpected error, SQLCODE is: ', SQLCA.SQLCODE) (skip, a, f(9));
        CALL SYS$PUTMSG(RDB$MESSAGE_VECTOR, 0, 0, 0);
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK;
        RETURN;
END CURSOR_EX;
```

♦

Example 2: Using the DESCRIBE . . . SELECT LIST statement to determine if a statement is a SELECT statement

```
SELECT_OR_NOT: PROCEDURE OPTIONS (MAIN);
/*
* Use the DESCRIBE...SELECT LIST statement to determine whether or not
* a dynamically generated statement is a SELECT statement.
*
* The program accepts a string from the terminal input that
* must be a valid SQL statement and then indicates whether
* it is a SELECT statement or not.
*/
declare sys$putmsg external entry (any, any value, any value, any value);

/* Declare SQL Communications Area: */
EXEC SQL INCLUDE SQLCA;
/* Declare SQL Descriptor Area: */
EXEC SQL INCLUDE SQLDA;

/*
 * Branch to ERR_HANDLER if the SQLCODE field
 * of the SQLCA is greater than 0:
 */
EXEC SQL WHENEVER SQLERROR GOTO ERR_HANDLER;

DECLARE STATEMENT_STRING CHAR(256);
GET EDIT (statement_string) (A(256))
        OPTIONS(PROMPT('Enter a valid SQL statement: '));
```

```
/* Allocate memory for the SQLDA and
 * set the value of its SQLN field:
 */
SQLSIZE = 10;
ALLOCATE SQLDA SET (SQLDAPTR);
SQLN = 10;

/* Prepare the statement string
 * and write information about any select-list
 * items to the SQLDA:
 */
EXEC SQL PREPARE prepared_statement
        FROM statement_string;

EXEC SQL DESCRIBE prepared_statement
        SELECT LIST INTO SQLDA;

/*
 * If the second element of SQLERRD array of the SQLCA is
 * one, we know the statement is a SELECT statement:
 */
IF SQLCA.SQLERRD(2) = 1 THEN
        PUT EDIT ('A SELECT statement') (A);
        ELSE
        PUT EDIT ('Not a SELECT statement') (A);

RETURN;

ERR_HANDLER:
        PUT EDIT
        ('Unexpected error, SQLCODE is: ', SQLCA.SQLCODE) (skip, a, f(9));
        CALL SYS$PUTMSG(RDB$MESSAGE_VECTOR, 0, 0, 0);
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        EXEC SQL ROLLBACK;
        RETURN;
END SELECT_OR_NOT;
```

## DISCONNECT Statement

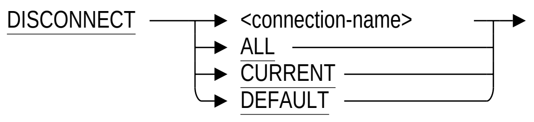
Detaches from declared databases and releases the aliases that you specified in the declarations. The DISCONNECT statement also ends the specified transactions and undoes all the changes you made since those transactions began.

### Environment

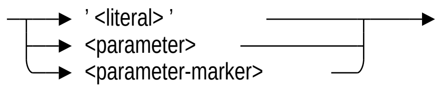
You can use the DISCONNECT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



connection-name =



### Arguments

**connection-name**
Specifies a name for the association between the group of databases being attached (the environment) and the database and request handles that reference them (the session).

You can specify the connection name as the following:

- A string literal enclosed in single quotation marks
- A parameter (in module language)
- A variable (in precompiled SQL)

**ALL**
Specifies all active connections.

**DEFAULT**
Specifies the default connection.

**CURRENT**
Specifies the current connection.

## Usage Notes

- Use the DISCONNECT DEFAULT statement instead of the FINISH statement. The FINISH statement is deprecated syntax. Because the DISCONNECT DEFAULT statement performs an automatic rollback, be sure to commit any changes that you want to keep before you execute the DISCONNECT statement.

  See the *Oracle Rdb7 Guide to SQL Programming* for disconnect information with module language procedures.

OpenVMS VAX OpenVMS Alpha

- You can use SQL connections and explicit calls to DECdtm services to control when you attach and detach from specific databases. By explicitly calling DECdtm system services and associating each database with an SQL connection, you can detach from one database while remaining attached to other databases. For more information, see the *Oracle Rdb7 Guide to Distributed Transactions*. ♦

## Examples

Example 1: Using the DISCONNECT statement in interactive SQL

This example in interactive SQL illustrates that the DISCONNECT statement lets you attach a database with the same alias as a previously attached database (in this example the alias is the default). Use the SHOW DATABASE statement to see the database settings.

```
SQL> ATTACH 'FILENAME mypers';
SQL> --
SQL> ATTACH 'FILENAME mypers';
This alias has already been declared.
Would you like to override this declaration (No)? no
%SQL-F-DEFDBDEC, A database has already been declared with the default alias
SQL> DISCONNECT DEFAULT;
SQL> ATTACH 'FILENAME mypers';
```

Example 2: Using the DISCONNECT statement in precompiled SQL

This example is taken from the sample program sql_connections.sc. To use connections in a program, you must specify the SQLOPTIONS=(CONNECT) or `-s 'conn'` qualifier on the precompiler command line. This example shows an EXEC SQL DISCONNECT statement that specifies the string literal 'a1' for the connection-name and EXEC SQL DISCONNECT statements that specify the keywords ALL and DEFAULT.

```
#include <stdio.h>
#include <string.h>
#include <descrip.h>

char   employee_id1[6];
char   last_name1[16];
char   employee_id2[16];
char   degree[14];
char   employee_id3[16];
char   supervisor[6];
char   employee_id4[6];
char   last_name4[15];

void sys$putmsg();

 EXEC SQL INCLUDE SQLCA;
 EXEC SQL declare         alias filename personnel;
 EXEC SQL declare alias_1 alias filename personnel;
 EXEC SQL declare alias_2 alias filename personnel;
 EXEC SQL declare alias_3 alias filename personnel;
 main()

 {
  printf("\n\n\n******* Disconnect from default ***************\n");
  EXEC SQL disconnect default;
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
 printf("\n\n\n");

 printf("********* Establish CONNECTION 1 **********\n");

 EXEC SQL connect to 'alias alias_1 filename personnel' as 'a1';
 if (SQLCA.SQLCODE !=  0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

 printf("********* Insert a record  **********\n");
```

```
EXEC SQL insert into alias_1.employees (employee_id, last_name)
values ('00301','FELDMAN');
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("********* Retrieve the record  **********\n");
EXEC SQL select employee_id, last_name into :employee_id1,
:last_name1 from alias_1.employees where employee_id = '00301';
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n\n");
printf ("Employee_id = %s\n",employee_id1);
printf ("Last_name   = %s\n",last_name1);
printf("\n\n\n");

printf("********* Establish CONNECTION 2 **********\n");
EXEC SQL connect to 'alias alias_2 filename personnel' as 'a2';
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("********* Insert a record  **********\n");
EXEC SQL insert into alias_2.degrees (employee_id, degree_field)
values ('00901','MASTERS');
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("********* Retrieve the record  **********\n");
EXEC SQL select employee_id, degree_field into :employee_id2,
:degree from alias_2.degrees where employee_id = '00901';
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n\n");
printf("Employee-id  = %s\n",employee_id2);
printf("Degree      = %s\n",degree);
printf("\n\n\n");

printf("********* Establish CONNECTION 3 **********\n");
EXEC SQL connect to 'alias alias_3 filename personnel' as 'a3';
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("********* Insert a record  **********\n");
EXEC SQL insert into alias_3.job_history (employee_id, supervisor_id)
values ('01501','Brown');
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

printf("********* Retrieve the record  **********\n");
EXEC SQL select employee_id, supervisor_id into :employee_id3,
:supervisor from alias_3.job_history where employee_id = '01501';
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
printf("\n\n\n");
printf("Employee-id  = %s\n",employee_id3);
printf("Supervisor   = %s\n ",supervisor);
printf("\n\n\n");

printf("********* Establish CONNECTION DEFAULT **********\n");
EXEC SQL set connect default ;
 if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
 printf("********* Retrieve record with id 00164 **********\n");
 EXEC SQL select employee_id, last_name into :employee_id4,
 :last_name4 from employees where employee_id = '00164';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
 printf("\n\n\n");
 printf("Employee_id  = %s\n",employee_id4);
 printf("Last_name    = %s\n",last_name4);
 printf("\n\n\n");

 printf("**** DISCONNECT, RECONNECT & TRY TO FIND RECORD *****\n");
 strcpy(employee_id1,"     ");
 strcpy(last_name1,"              ");
 EXEC SQL disconnect 'a1';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
 EXEC SQL connect to 'alias alias_1 filename personnel' as 'a1';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
 printf("********* Retrieve the record  **********\n");
 EXEC SQL select employee_id, last_name into :employee_id1,
 :last_name1 from alias_1.employees where employee_id = '00301';
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);

 printf("************** SHOULD DISPLAY NO RECORD **************\n");
 printf("\n\n\n");
 printf("employee_id = %s\n",employee_id1);
 printf("last_name   = %s\n",last_name1);
 printf("\n\n\n");
  printf("*************** DISCONNECT ALL CONNECTIONS ***************\n");
 EXEC SQL disconnect all;
  if (SQLCA.SQLCODE != 0) SYS$PUTMSG(&RDB$MESSAGE_VECTOR,0,0,0);
 EXEC SQL rollback;
}
```

## DROP CATALOG Statement

Deletes the specified catalog definition. You must delete all schemas and definitions contained in a catalog before you can delete that catalog. If other definitions exist that refer to the named catalog, the deletion fails.

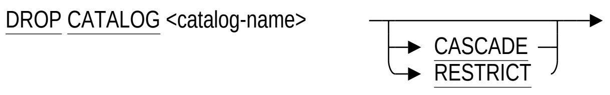
The DROP CATALOG statement lists all schemas and definitions that it is going to delete. You can roll back the statement if you do not want to delete these definitions.

### Environment

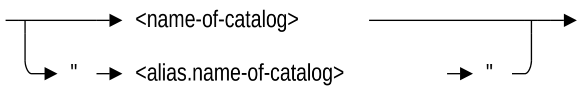
You can use the DROP CATALOG statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



catalog-name =



### Arguments

**DROP CATALOG catalog-name CASCADE**
**DROP CATALOG catalog-name RESTRICT**
Performs a restricted delete by default. If you prefer to delete all definitions contained in the catalog, you can specify the DROP CATALOG CASCADE statement.

**catalog-name**
Specifies the module catalog name.

**alias.name-of-catalog**
Specifies a name for the attachment to the database. Always qualify the catalog name with an alias if your program or interactive SQL statements refer to more than one database. Separate the name of the catalog from the alias with a period, and enclose the qualified name in double quotation marks.

You must issue a SET QUOTING RULES statement before you can qualify a catalog name with an alias.

## Usage Notes

- You cannot delete a catalog if another user issued a query using that catalog. Users must detach from the database with a DISCONNECT statement before you can delete the catalog.
- You cannot delete the catalog RDB$CATALOG.

## Example

Example 1: Deleting a catalog

The following statement deletes the catalog DEPT1 associated with the alias PERSONNEL_ALIAS:

```
SQL> ATTACH 'ALIAS PERSONNEL_ALIAS FILENAME CORPORATE_DATA';
SQL> SET QUOTING RULES 'SQL92';
SQL> CREATE CATALOG "PERSONNEL_ALIAS.DEPT1";
SQL> SHOW CATALOG;
Catalogs in database PERSONNEL_ALIAS
    "PERSONNEL_ALIAS.ADMINISTRATION"
    "PERSONNEL_ALIAS.RDB$CATALOG""
    "PERSONNEL_ALIAS.DEPT1"
SQL> DROP CATALOG "PERSONNEL_ALIAS.DEPT1";
SQL> SHOW CATALOG;
Catalogs in database PERSONNEL_ALIAS
    "PERSONNEL_ALIAS.ADMINISTRATION"
    "PERSONNEL_ALIAS.RDB$CATALOG"
SQL> DROP CATALOG "PERSONNEL_ALIAS.RDB$CATALOG";
%SQL-F-NODROPSYSCAT, Catalog "PERSONNEL_ALIAS.RDB$CATALOG" may not be
dropped
SQL>
```

## DROP COLLATING SEQUENCE Statement

Deletes the named collating sequence.

You cannot delete a collating sequence if it is used by the database or by any domain in the database.

### Environment

You can use the DROP COLLATING SEQUENCE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP COLLATING SEQUENCE  ──► <sequence-name>  ──►
```

### Arguments

**sequence-name**
Specifies the sequence-name argument you used when creating the collating sequence in the CREATE COLLATING SEQUENCE statement.

### Usage Notes

- You must execute this statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL starts a read/write transaction implicitly.
- Other users are allowed to be attached to the database when you issue the DROP COLLATING SEQUENCE statement.

## Examples

Example 1: Creating, then deleting, a French collating sequence

The following example creates a collating sequence using the predefined collating sequence FRENCH. It then uses the SHOW COLLATING SEQUENCE statement to show the defined collating sequence.

The example next deletes the collating sequence using the DROP COLLATING SEQUENCE statement. The SHOW COLLATING SEQUENCE statement shows that the collating sequence no longer exists.

```
SQL> ATTACH 'FILENAME personnel';
SQL> CREATE COLLATING SEQUENCE FRENCH FRENCH;
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
     FRENCH
SQL> --
SQL> DROP COLLATING SEQUENCE FRENCH;
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
No collating sequences found
```

Example 2: Deleting a collating sequence used to define a domain or database

The following example shows that you cannot delete a collating sequence if a domain or database is defined using the collating sequence:

```
SQL> CREATE COLLATING SEQUENCE SPANISH SPANISH;
SQL> CREATE DOMAIN LAST_NAME_SPANISH CHAR (14)
cont> COLLATING SEQUENCE IS SPANISH;
SQL> --
SQL> SHOW DOMAIN LAST_NAME_SPANISH
LAST_NAME_SPANISH               CHAR(14)
 Collating sequence: SPANISH
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
     SPANISH
SQL> --
SQL> -- You cannot delete the collating sequence because the
SQL> -- domain LAST_NAME_SPANISH, defined using SPANISH, still exists:
SQL> --
SQL> DROP COLLATING SEQUENCE SPANISH;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-COLUSEDFLD, the collating sequence named SPANISH is used in
field LAST_NAME_SPANISH
```

```
SQL> --
SQL> -- Delete the domain:
SQL> --
SQL> DROP DOMAIN LAST_NAME_SPANISH;
SQL> --
SQL> -- Now you can delete the collating sequence:
SQL> --
SQL> DROP COLLATING SEQUENCE SPANISH;
SQL> --
SQL> SHOW COLLATING SEQUENCE
User collating sequences in database with filename personnel
No collating sequences found
```

## DROP CONSTRAINT Statement

Deletes constraints defined with versions of the SQL interface provided prior to Version 3.1 of Oracle Rdb.

### Environment

You can use the DROP CONSTRAINT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP CONSTRAINT ──► <constraint-name> ──►
```

### Arguments

**constraint-name**
Specifies the name of the constraint that you want to delete.

### Usage Notes

- Attempts to delete a constraint fail if that constraint is in a table involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the constraint. When Oracle Rdb first accesses an object such as the constraint, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' access to the object.

  Similarly, while you are deleting a constraint, users cannot execute queries involving that constraint until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user will receive a LOCK CONFLICT ON CLIENT error message. While DDL operations are performed, normal data locking mechanisms are used against system tables. (System tables contain information about objects in the database.) Therefore, attempts to update an object lock out attempts to query that object. These locks are held until the commit or rollback of the DDL operation.

The WAIT/NOWAIT clause of the SET TRANSACTION statement does not affect attempts to update metadata with simultaneous queries. Even if you specify SET TRANSACTION WAIT for the metadata update transaction, you get the following error message if a lock conflict exists:

```
%RDB-E-LOCK_CONFLICT, request failed due to locked resource; no-wait
parameter specified for transaction
-RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-LCKCNFLCT, lock conflict on client
SQL>
```

However, a user's query waits for a metadata update to complete with a ROLLBACK or COMMIT statement, even if the user specified NOWAIT in the SET TRANSACTION statement.

- Table-specific constraints defined with the CREATE TABLE statement of the SQL interface provided with Version 3.1 or higher of Oracle Rdb cannot be deleted using the DROP CONSTRAINT statement.

  Use the DROP CONSTRAINT clause of the ALTER TABLE statement to delete this type of constraint.
- You cannot execute the DROP CONSTRAINT statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.

## Example

Example 1: Deleting a constraint defined with a previous version of SQL

The following statement deletes the SEX_NOT_NULL constraint defined with a version of the SQL interface provided prior to Version 3.1 of Oracle Rdb:

```
SQL> DROP CONSTRAINT SEX_NOT_NULL;
```

## DROP DATABASE Statement

Deletes a database.

When this statement executes in Oracle Rdb, SQL deletes all the database root and storage area files associated with the database.

OpenVMS VAX OpenVMS Alpha If you specify a repository path name in the DROP DATABASE statement or specify an alias for a database attached with the PATHNAME argument, SQL also deletes the repository directory that contains the database definitions. ♦

**Caution**

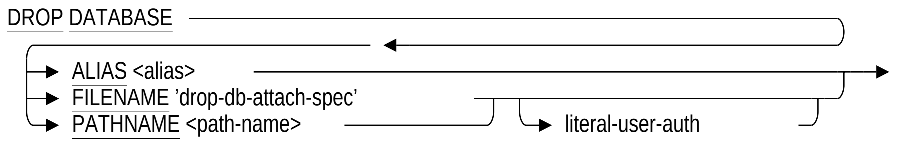
Use the DROP DATABASE statement with care. You cannot use the ROLLBACK statement to cancel a DROP DATABASE statement. When you use this statement, SQL deletes the database root and storage area files, which include all data and all definitions.

### Environment

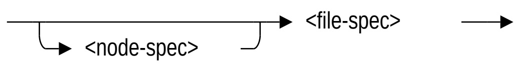
You can use the DROP DATABASE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

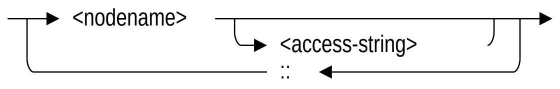


drop-db-attach-spec =

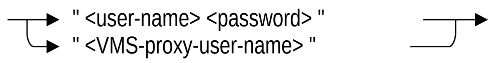


## DROP DATABASE Statement

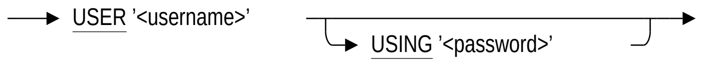
node-spec =



access-string =



literal-user-auth =



## Arguments

**ALIAS alias**

Specifies the alias for an attached database. The DROP DATABASE statement deletes the database and all database root and storage area files associated with the alias.

OpenVMS VAX OpenVMS Alpha If the database was declared with the PATHNAME argument, the DROP DATABASE statement also deletes the repository directory that contains the database definitions. ♦

**FILENAME 'attach-spec'**

Specifies a quoted string containing full or partial information needed to access a database.

For an Oracle Rdb database, an attach specification contains the file specification of the .rdb file.

For Oracle Rdb, the DROP DATABASE statement deletes the database and all database system files associated with the database root file specification. If you use a partial file specification, SQL uses the standard defaults. If you use a full file specification, the DROP DATABASE statement deletes only the database files, whether or not there is also a repository directory containing database definitions. Because the DROP DATABASE statement may delete more than one file with different file extensions, do not specify a file extension with the file specification.

OpenVMS VAX OpenVMS Alpha

**PATHNAME path-name**
Specifies a full or relative repository path name for the repository directory where the database definitions are stored. Use a path name instead of a file specification to delete the repository database definitions from the repository along with the database root and storage area files.

The PATHNAME clause is available only on OpenVMS platforms. ♦

**literal-user-auth**
Specifies the user name and password for access to databases, particularly remote database.

This literal lets you explicitly provide user name and password information in the DROP DATABASE statement.

**USER ’username’**
Defines a character string literal that specifies the operating system user name that the database system uses for privilege checking.

**USING ’password’**
Defines a character string literal that specifies the user’s password for the user name specified in the USER clause.

## Usage Notes

- You cannot delete an Oracle Rdb database when other users have declared it. This means you cannot issue a DROP DATABASE statement for an Oracle Rdb database after another user attaches to that database.
- The CASCADE and RESTRICT keywords are not allowed on the DROP DATABASE statement. The DROP DATABASE statement has only one mode of behavior; it always performs an implicit *cascading delete*.

## Examples

Example 1: Deleting files only

The following statement deletes the database system files for the database associated with the database personnel.rdb. If this database also had definitions stored in a repository directory, this DROP DATABASE statement would not delete those definitions.

```
SQL> DROP DATABASE FILENAME personnel;
```

OpenVMS VAX OpenVMS Alpha

Example 2: Deleting files and repository definitions

To delete database files and repository definitions, you must specify a repository path name in the DROP DATABASE statement. This statement deletes the repository directory CDD$TOP.ACCOUNTING.PERSONNEL in addition to all database root and storage area files associated with it.

```
SQL> DROP DATABASE PATHNAME CDD$TOP.ACCOUNTING.PERSONNEL;
```
♦

## DROP DOMAIN Statement

Deletes a domain definition. If you attached to the database using the PATHNAME qualifier, SQL also deletes the domain definition from the repository.

### Environment

You can use the DROP DOMAIN statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP DOMAIN ──► <domain-name> ───►
```

### Arguments

**domain-name**
Specifies the name of the domain you want to delete.

### Usage Notes

- You can delete any named domain. However, you cannot delete a domain that is referred to in a column definition in a table. If you want to delete a domain that is referred to in a column definition, you must first use the ALTER TABLE statement to delete the column definition. If the column definition is used in a constraint or index definition, you must first delete the constraint or index definition, then delete the column definition.
- You cannot delete a domain definition unless you attach to the database that includes the domain.

- You must execute the DROP DOMAIN statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- The DROP DOMAIN statement fails when the following are true:
  - The database to which it applies was created with the DICTIONARY IS REQUIRED argument.
  - The database was attached using the FILENAME argument.

  Under these circumstances, the statement fails with the following error when you issue it:

  ```
  %RDB-E-NO_META_UPDATE, metadata update failed
  -RDMS-F-CDDISREQ, CDD required for metadata updates
                    is not being maintained
  ```
- Because Oracle Rdb creates dependencies between stored routines and metadata (like domains) on which they are compiled and stored, you cannot delete a domain with a routine dependency. Because the DROP DOMAIN statement fails when you attempt to delete a domain referenced in a stored routine, the dependent stored routine is not invalidated. Refer to the CREATE MODULE Statement for a list of statements that can or cannot cause stored routine invalidation.

  Refer to the *Oracle Rdb7 Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.
- You cannot execute the DROP DOMAIN statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- If a domain is deleted as part of a DROP SCHEMA CASCADE statement, the domain properties are inherited by any columns defined using the domain.

## Examples

Example 1: Deleting a domain not referred to by columns

```
SQL> --
SQL> -- The following CREATE DOMAIN statement creates a domain
SQL> -- that is not used by any columns:
SQL> --
SQL> CREATE DOMAIN ABCD IS CHAR(4);
SQL> --
SQL> -- The SHOW DOMAINS statement shows domain ABCD at the
SQL> -- top of the list:
SQL> --
SQL> SHOW DOMAINS

User domains in database with filename personnel
ABCD                            CHAR(4)
ADDRESS_DATA_1_DOM              CHAR(25)
ADDRESS_DATA_2_DOM              CHAR(20)
   .
   .
   .

SQL> --
SQL> -- Now delete the domain:
SQL> --
SQL> DROP DOMAIN ABCD;
SQL> --
SQL> -- The SHOW DOMAINS statement shows that the
SQL> -- domain ABCD has been deleted:
SQL> --
SQL> SHOW DOMAINS

User domains in database with filename personnel
ADDRESS_DATA_1_DOM              CHAR(25)
ADDRESS_DATA_2_DOM              CHAR(20)
   .
   .
   .
```

Example 2: Deleting a domain referred to by columns

The following example deletes a domain definition. Because a column refers to the domain definition and a constraint refers to the column, you must first alter the table before deleting the domain.

```
SQL> --
SQL> -- Attempt to delete the domain SEX_DOM.  Error messages
SQL> -- indicate that the table EMPLOYEES uses the domain
SQL> -- SEX_DOM, so SEX_DOM cannot yet be deleted:
SQL> --
SQL> DROP DOMAIN SEX_DOM;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELEXI, field SEX_DOM is used in relation EMPLOYEES
-RDMS-F-FLDNOTDEL, field SEX_DOM has not been deleted
SQL> --
SQL> -- Looking at the EMPLOYEES table shows that SEX is the
SQL> -- column that depends on the domain SEX_DOM.  Try
SQL> -- to delete the column SEX; error messages indicate that the
SQL> -- constraint EMP_SEX_VALUES depends on the column SEX:
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP COLUMN SEX;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-FLDINCON, field SEX is referenced in constraint EMP_SEX_VALUES
-RDMS-F-RELFLDNOD, field SEX has not been deleted
from relation EMPLOYEES
SQL> --
SQL> -- Delete the constraint EMP_SEX_VALUES:
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP CONSTRAINT EMP_SEX_VALUES;
SQL> --
SQL> -- Because EMP_SEX_VALUES was the only constraint or index
SQL> -- that depended on the column SEX, you can now delete
SQL> -- the column SEX:
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP COLUMN SEX;
SQL> --
SQL> -- The column SEX in the table EMPLOYEES was the only column in
SQL> -- the database that depended on the domain SEX_DOM, so you can
SQL> -- now delete the domain SEX_DOM:
SQL> --
SQL> DROP DOMAIN SEX_DOM;
SQL>
```

## DROP INDEX Statement

Deletes the specified index definition. If you attach to the database using the PATHNAME qualifier, SQL also deletes the index definition from the repository.

### Environment

You can use the DROP INDEX statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP INDEX ──► <index-name> ──►
```

### Arguments

**index-name**
Specifies the name of the index definition you want to delete.

### Usage Notes

- You cannot delete an index definition unless you are attached to the database that includes the index.
- You must execute this statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- Attempts to delete an index fail if that index is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the index. When Oracle Rdb first accesses an object such as the index, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object,

you get a LOCK CONFLICT ON CLIENT message due to the other users' optimized access to the object.

Similarly, while you are deleting an index, users cannot execute queries involving that index until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user receives a LOCK CONFLICT ON CLIENT error message. While DDL operations are performed, normal data locking mechanisms are used against system tables. (System tables contain information about objects in the database.) Therefore, attempts to update an object lock out attempts to query that object. These locks are held until the COMMIT or ROLLBACK statement is issued for the DDL operation.

The WAIT/NOWAIT clause of the SET TRANSACTION statement does not affect attempts to update metadata with simultaneous queries. Even if you specify SET TRANSACTION WAIT for the metadata update transaction, you get the following error message if a lock conflict exists:

```
%RDB-E-LOCK_CONFLICT, request failed due to locked resource; no-wait
parameter specified for transaction
-RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-LCKCNFLCT, lock conflict on client
SQL>
```

However, a user's query waits for a metadata update to complete with a ROLLBACK or COMMIT statement, even if the user specified NOWAIT in the SET TRANSACTION statement.

- The DROP INDEX statement fails when the following are true:
  - The database to which it applies was created with the DICTIONARY IS REQUIRED argument.
  - The database was attached using the FILENAME argument.

  Under these circumstances, the statement fails with the following error when you issue it:

```
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CDDISREQ, CDD required for metadata updates is not being maintained
```

- You cannot execute the DROP INDEX statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- The DROP INDEX statement marks any query outline associated with the deleted index as invalid.

- You cannot delete an index that is used in a storage map placement clause. You must alter the storage map using the NO PLACEMENT VIA INDEX clause before you can delete the index.

## Examples

Example 1: Deleting an index from the default database

```
SQL> ATTACH 'FILENAME personnel';
SQL> DROP INDEX DEG_COLLEGE_CODE;
SQL> COMMIT;
```

Example 2: Deleting an index from one of several attached databases

```
SQL> ATTACH 'FILENAME personnel';
SQL> ATTACH 'ALIAS MF FILENAME mf_personnel';
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> SET QUOTING RULES 'SQL92';
SQL> DROP INDEX "CORP.ADMINISTRATION".PERSONNEL.EMP_EMPLOYEE_ID;
SQL> COMMIT;
```

## DROP MODULE Statement

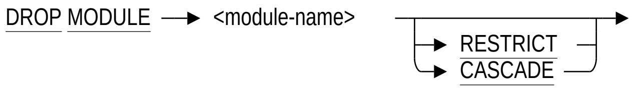
Deletes a module from an Oracle Rdb database.

### Environment

You can use the DROP MODULE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**module-name**
Identifies the name of the module.

**RESTRICT**
Prevents the removal of a stored routine definition when the function or procedure is referenced by any other object within an Oracle Rdb database. RESTRICT is the default.

**CASCADE**
Specifies that you want SQL to invalidate all objects that refer to routines in the module and then delete that module definition. This is known as a cascading delete. If you delete a module referenced by a stored routine with a routine or language-semantic dependency, SQL also marks the affected stored routine as invalid.

## Usage Notes

- To execute this statement, you need the DROP privilege on the module you want to delete.
- Attempts to delete a module will fail if the objects in a procedure or function of a stored module are involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the module. When SQL first accesses an object such as a module, a lock is placed on that object and not released until the users exit the database.

  If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' access to the object.

  Similarly, while you are deleting a module, users cannot execute queries involving the procedure or function of a module until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user receives a LOCK CONFLICT ON CLIENT error message. While data definition operations are performed, normal data locking mechanisms are used against system tables. (System tables contain information about objects in the database.) Therefore, attempts to update an object lock out attempts to query that object. These locks are held until the commit or rollback of the data definition operation.

  However, a user's query waits for a metadata update to complete with a ROLLBACK or COMMIT statement, even if the user specified NOWAIT in the SET TRANSACTION statement.
- You must execute the DROP MODULE statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- You cannot execute the DROP MODULE statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- If a table has a computed-by column whose definition invokes a stored function, and if that stored function has previously been deleted, the column is set to NULL.

- If a module is deleted, invalidating a stored routine, and then the module is redefined, use of the invalid routine causes attempts to revalidate the routine references.

  You can define the RDMS$VALIDATE_ROUTINE logical name or RDB_VALIDATE_ROUTINE configuration parameter to the value of "1" and, if the routine is again valid after re-creating the objects referenced, then Oracle Rdb clears the invalid setting from the database. This is useful if you need to delete a widely used module to correct an algorithmic change and leave the structure of the routine the same.

## Examples

Example 1: Removing a module from an Oracle Rdb database

```
SQL> DROP MODULE employee_salary;
```

## DROP OUTLINE Statement

Deletes a query outline.

### Environment

You can use the DROP OUTLINE statement only in interactive SQL.

The DROP OUTLINE statement allows the user to specify that an existing outline should be removed from the database.

### Format

```
DROP OUTLINE ──▶ <outline-name> ──▶
```

### Arguments

**outline-name**
Specifies the name of the outline you want to delete.

### Usage Notes

- To delete an outline, you must have the SQL DROP privilege for every table referenced by the outline.
- The DROP OUTLINE statement is an online operation (other users can be attached to the database when an outline is deleted). However, a query outline cannot be deleted when the outline is being referenced in another transaction. If you issue a DROP OUTLINE statement while another transaction is referencing the outline, the transaction finishes and then the outline is deleted.

### Examples

Example 1. Deleting an outline

```
SQL> DROP OUTLINE MY_OUTLINE;
```

## DROP PATHNAME Statement

OpenVMS VAX OpenVMS Alpha Deletes the repository definitions. It does not delete the physical database files.

This statement can be used only on OpenVMS platforms.

### Environment

You can use the DROP PATHNAME statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP PATHNAME ──► <path-name> ──►
```

### Arguments

**path-name**
Specifies the repository path name for the schema definitions.

Specify either a full path name or a relative path name. If you use a relative path name, the current default repository directory must be defined to include all the path name segments that precede the relative path name.

### Examples

Example 1: Deleting a path name with the DROP PATHNAME statement

The following example deletes CDD$TOP.SQL.DEPT3, a repository directory, and all its descendants. It does not delete the database system files or data that corresponds to that path name.

```
SQL> DROP PATHNAME "CDD$TOP.SQL.DEPT3";
```

♦

## Drop Routine Statement

Deletes a routine definition, external or stored, from an Oracle Rdb database. **External routine** refers to both external functions and external procedures. **Stored routine** refers to both stored functions and stored procedures.

If you specify RESTRICT when you delete a routine definition, SQL executes the delete operation only if the routine is not referenced by any other object in the database.

If you specify CASCADE when you delete a routine definition, SQL executes the delete and invalidates any referencing routines.

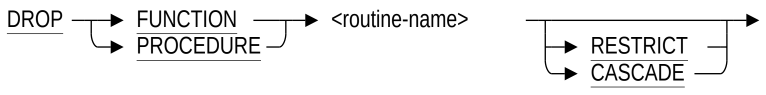
Because you cannot alter a routine definition, you must first delete the external routine or the module containing the stored routine and then apply the required changes to a new external routine or module definition.

### Environment

You can use the DROP FUNCTION and DROP PROCEDURE statements:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**DROP FUNCTION routine-name**
Identifies the name of the external or stored function definition in the Oracle Rdb database.

**DROP PROCEDURE routine-name**
Identifies the name of the external or stored procedure definition in the Oracle Rdb database.

**RESTRICT**
Prevents the removal of an external or stored routine definition when the routine is referenced by any other object within an Oracle Rdb database.

RESTRICT is the default.

**CASCADE**
Deletes the routine definition even when there are dependencies on the specified routine. Any referencing routines are marked invalid.

## Usage Notes

- SQL does not store the external routine's executable image in an Oracle Rdb database. Instead, it stores information that points to the external routine, such as its name and location, so that SQL can automatically invoke it from within a query execution.
- Before you can delete a stored routine, you must have ALTER privileges on the module containing the stored routine that you want to delete.
- Computed-by columns are set to NULL in tables that reference a function that has been deleted by a DROP FUNCTION CASCADE statement.

  You can alter the table and delete the computed-by column. At some future point, you can then alter the table and create a new computed-by column using the same name but with a different computed-by expression.
- To delete an external routine you must have DROP privileges.
- You cannot execute the DROP FUNCTION or DROP PROCEDURE statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.

## Examples

Example 1: Deleting an external function definition from an Oracle Rdb database

If you want to alter an external function definition, you must first delete it and then create it again with the changes you plan. This example shows how to delete the COSINE_F function.

```
SQL> DROP FUNCTION cosine_f RESTRICT;
```

Example 2: Deleting a stored function

```
SQL> DROP FUNCTION abs;
```

## DROP SCHEMA Statement

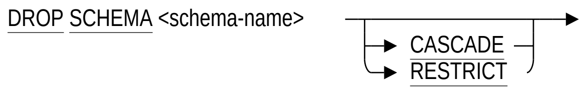
Deletes a schema and all the definitions that it contains.

### Environment

You can use the DROP SCHEMA statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**schema-name**
Specifies the module schema name. You must qualify the schema name with catalog and alias names if the schema is not in the default catalog and database. For more information about schema names, see Section 2.2.8.

**CASCADE**
Deletes all other definitions (views, constraints, indexes, and triggers) that refer to the named schema and then deletes that schema definition. This is known as a cascading delete.

If you specify the CASCADE keyword, SQL deletes all definitions contained by the schema before deleting the schema.

If you do not specify the CASCADE keyword, the schema must be empty.

**RESTRICT**
Returns an error message if other definitions refer to the named schema. The DROP SCHEMA RESTRICT statement will not delete a schema until you have deleted all other definitions that refer to the named schema. The DROP SCHEMA statement specifies an implicit RESTRICT by default.

## Usage Notes

- If you try to delete a schema without first deleting views, constraints, indexes, and triggers that refer to that schema, SQL issues the following error message:

  ```
  SQL> ATTACH 'ALIAS MS_ALIAS FILENAME MS_TESTDB';
  SQL> SET QUOTING RULES 'SQL92';
  SQL> SET CATALOG '"MS_ALIAS.MS_TESTCATALOG"';
  SQL> DROP SCHEMA "MS_ALIAS.MS_TESTSCHEMA";
  %RDB-E-NO_META_UPDATE, metadata update failed
  -RDMS-E-SCHEMAINUSE, schema MS_TESTSCHEMA currently in use
  ```

  You can avoid the error message by deleting all the definitions that refer to the named schema before deleting the schema, or by specifying DROP SCHEMA CASCADE.
- To delete a schema, you must have the same authorization identifier as that schema or your user name must match the schema name.
- You cannot delete the schema RDB$SCHEMA.

## Examples

Example 1: Deleting a schema with implicit RESTRICT

In the following example, the user must delete the definitions that refer to the schema RECRUITING before deleting the schema itself.

After setting the default schema to RECRUITING and the default catalog to ADMINISTRATION, the user can qualify each definition name with only the alias CORP.

```
SQL> ATTACH 'ALIAS CORP FILENAME CORPORATE_DATA';
SQL> SET CATALOG '"CORP.ADMINISTRATION"';
SQL> SET SCHEMA '"CORP.ADMINISTRATION".RECRUITING';
SQL> SET QUOTING RULES 'SQL92';
SQL> DROP SCHEMA "CORP.RECRUITING";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-SCHEMAINUSE, schema RECRUITING currently in use
SQL> DROP TABLE "CORP.CANDIDATES";
SQL> DROP TABLE "CORP.COLLEGES";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CONEXI, relation COLLEGES is referenced in constraint DEGREES_FOREIGN3
-RDMS-F-RELNOTDEL, relation COLLEGES has not been deleted
SQL> DROP TABLE "CORP.DEGREES";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-TRGEXI, relation DEGREES is referenced in trigger
EMPLOYEE_ID_CASCADE_DELETE
SQL> DROP TABLE "CORP.RESUMES";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-TRGEXI, relation RESUMES is referenced in trigger
EMPLOYEE_ID_CASCADE_DELETE
-RDMS-F-RELNOTDEL, relation RESUMES has not been deleted
SQL> --
SQL> -- The trigger is part of another schema, PERSONNEL. Since this
SQL> -- is not the default schema, the user qualifies the trigger name
SQL> -- with schema and names.
SQL> --
SQL> DROP TRIGGER "CORP.ADMINSTRATION".PERSONNEL.EMPLOYEE_ID_CASCADE_DELETE;
SQL> DROP CONSTRAINT "CORP.DEGREES_FOREIGN3";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CONDELVIAREL, constraint DEGREES_FOREIGN3 can only be deleted by
changing or deleting relation DEGREES
SQL> DROP TABLE "CORP.DEGREES";
SQL> DROP TABLE "CORP.RESUMES";
SQL> DROP TABLE "CORP.COLLEGES";
SQL> DROP SCHEMA "CORP.RECRUITING";
```

Example 2: Deleting a schema with CASCADE

In the following example, SQL deletes the definitions that refer to the schema ACCOUNTING, then deletes the schema itself:

```
SQL> DROP SCHEMA "CORP.ACCOUNTING" CASCADE;
Domain "CORP.ADMINISTRATION".ACCOUNTING.BUDGET is also being dropped.
Domain "CORP.ADMINISTRATION".ACCOUNTING.CODE is also being dropped.
SQL>
```

## DROP STORAGE MAP Statement

Deletes the specified storage map definition.

### Environment

You can use the DROP STORAGE MAP statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP STORAGE MAP ──► <map-name> ──►
```

### Arguments

**map-name**
Specifies the name of the storage map you want to delete.

### Usage Notes

- You cannot delete a storage map that refers to a table that contains data. If you attempt to do so, you receive an error message. However, you can delete the table once the necessary views and constraints have been deleted. The underlying storage map is deleted with the table and its data.
- You must execute this statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- Attempts to delete a storage map fail if that storage map refers to a table that is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the storage map. When Oracle Rdb first accesses an object such as the storage map, a lock is placed on that object and not released until the users exit the

database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' access to the object.

Similarly, while you are deleting a storage map, users cannot execute queries involving tables that the storage map refers to until you complete the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The users receive a LOCK CONFLICT ON CLIENT error message. While DDL operations are performed, normal data locking mechanisms are used against system tables. (System tables contain information about objects in the database.) Therefore, attempts to update an object lock out attempts to query that object. These locks are held until the COMMIT or ROLLBACK of the DDL operation.

The WAIT/NOWAIT clause of the SET TRANSACTION statement does not affect attempts to update metadata with simultaneous queries. Even if you specify SET TRANSACTION WAIT for the metadata update transaction, you get the following error message if a lock conflict exists:

```
%RDB-E-LOCK_CONFLICT, request failed due to locked resource; no-wait
parameter specified for transaction
-RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-LCKCNFLCT, lock conflict on client
SQL>
```

However, a user's query waits for a metadata update to complete with a rollback or commit operation, even if the user specified NOWAIT in the SET TRANSACTION statement.

- The DROP STORAGE MAP statement fails when the following are true:
  - The database to which it applies was created with the DICTIONARY IS REQUIRED argument.
  - The database was attached using the FILENAME argument.

  Under these circumstances, the statement fails with the following error when you issue it:

  ```
  %RDB-E-NO_META_UPDATE, metadata update failed
  -RDMS-F-CDDISREQ, CDD required for metadata updates is not being maintained
  ```

- Other users are allowed to be attached to the database when you issue the DROP STORAGE MAP statement.
- You cannot execute the DROP STORAGE MAP statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.

## Examples

Example 1: Deleting a storage map in interactive SQL

This example deletes a storage map. You cannot delete a storage map that refers to a table that contains data.

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> DROP STORAGE MAP WORK_STATUS_MAP;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELNOTEMPTY, relation WORK_STATUS has data in it
SQL> DELETE FROM WORK_STATUS;
3 rows deleted
SQL> DROP STORAGE MAP WORK_STATUS_MAP;
SQL>
```

## DROP TABLE Statement

Deletes the specified table definition.

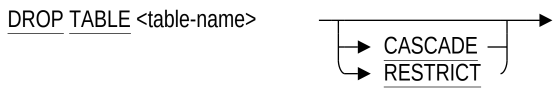
When the DROP TABLE statement executes, SQL first checks to ensure that there are no schema objects that reference the table. If there are no objects and you specify CASCADE, SQL deletes all other definitions that refer to the named table and then deletes that table definition from the database and the data stored in that table from the database. If you use the PATHNAME qualifier when you attach to the database, the DROP TABLE statement also deletes the table definition from the repository.

### Environment

You can use the DROP TABLE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**table-name**
Specifies the name of the table definition you want to delete.

**CASCADE**
Specifies that you want SQL to delete all other definitions (constraints, indexes, modules, storage maps, triggers, and views) that refer to the named table and then delete that table definition. This is known as a cascading delete. For modules, if you delete a table referenced by a stored routine with a routine or language-semantic dependency, SQL also marks the affected stored routines as invalid.

**RESTRICT**
Specifies that you want SQL to delete only the named table definition. If constraints, modules, triggers, or views are defined that refer to the named table, SQL does not delete the table. If there are indexes that refer to the named table, SQL deletes the table and does not issue an error.

For modules, because the DROP DOMAIN RESTRICT statement fails when you attempt to delete a table referenced in a stored routine, the dependent stored routine is not invalidated.

## Usage Notes

- You cannot delete a table when there are other active transactions involving the table. That is, you must have exclusive access to the table.
- If you do not specify either the CASCADE keyword or the RESTRICT keyword, SQL executes a restricted delete by default.
- Attempts to delete a table will fail if that table is involved in a query at the same time. You must detach from the database with a DISCONNECT statement before you can delete the table. When SQL first accesses an object such as the table, a lock is placed on that object and not released until the user exits the database.

  If you are using Oracle Rdb and attempt to update this object, you get a lock conflict on client message due to the other user's access to the object.

  Similarly, while you are deleting a table, users cannot execute queries involving that table until you completed the transaction with a COMMIT or ROLLBACK statement for the DROP statement. If you are using Oracle Rdb, users receive a lock conflict on client error message. While data definition language operations are performed, normal data locking mechanisms are used against system tables. (System tables contain information about objects in the database.) Therefore, attempts to update an object lock out attempts to query that object. These locks are held until the commit or rollback of the DDL operation.

  The WAIT/NOWAIT clause of the SET TRANSACTION statement does not affect attempts to update metadata with simultaneous queries. Even if you specify SET TRANSACTION WAIT for the metadata update transaction, you get the following error message if a lock conflict exists:

  ```
  %RDB-E-LOCK_CONFLICT, request failed due to locked resource
  -RDB-E-NO_META_UPDATE, metadata update failed
  -RDMS-F-LCKCNFLCT, lock conflict on client
  SQL>
  ```

However, a user's query will wait for a metadata update to complete with a ROLLBACK or COMMIT statement, even if the user specified NOWAIT in the SET TRANSACTION statement.

- You must execute the DROP TABLE statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- You cannot execute the DROP TABLE statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- Performance of DROP TABLE statements varies widely, depending on how the storage area file containing the table was defined. In multifile databases, performance is much slower if the base storage area was created with the PAGE FORMAT IS MIXED storage area parameter.
- You cannot delete a table definition unless you attached to the database that includes the table. Other users are allowed to be attached to the database when you issue the DROP TABLE statement.
- If a constraint in the database refers to a table, you cannot delete that table unless you either delete the constraint that refers to the table or use the DROP TABLE CASCADE statement. SQL returns the following error message:

```
SQL> DROP TABLE PERS.EMPLOYEES;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CONEXI, relation EMPLOYEES is referenced in constraint
RESUMES_FOREIGN1
-RDMS-F-RELNOTDEL, relation EMPLOYEES has not been deleted
```

- If a view definition refers to a table you want to delete, you must delete that view definition before you delete the table, or specify CASCADE.
- If a trigger definition refers to a table you want to delete, you must delete that trigger definition before you delete the table, or specify CASCADE.
- If an index definition refers to a table you want to delete, you do not need to delete that index definition before you delete the table.
- The DROP TABLE statement fails when the following are true:
  - The database to which it applies was created with the DICTIONARY IS REQUIRED argument.
  - The database was attached using the FILENAME argument.

Under these circumstances, the statement fails with the following error when you issue it:

```
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CDDISREQ, CDD required for metadata updates is not being maintained
```

- Because Oracle Rdb creates dependencies between stored routines and metadata (like tables) on which they are compiled and stored, you can delete a table with a routine or language-semantic dependency if you specify CASCADE but you cannot with RESTRICT. In the case of DROP TABLE CASCADE, when the table referenced in a stored routine is deleted, the stored routine is marked as invalid in the RDB$INTERRELATIONS system table. In the case of DROP TABLE RESTRICT, because the statement fails when you attempt to delete a table referenced in a stored routine, the dependent stored routine is not invalidated. See the CREATE MODULE Statement for a list of statements that can or cannot cause stored routine invalidation.

  See the *Oracle Rdb7 Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.

- The DROP TABLE statement marks any query outline that refers to the deleted table as invalid.

- A computed-by column is set to NULL if it references a persistent base table, global temporary table, or local temporary table that has been deleted by a DROP TABLE CASCADE statement. For example:

```
SQL> CREATE TABLE t1 (col1 INTEGER,
cont>                 col2 INTEGER);
SQL> --
SQL> CREATE TABLE t2 (x INTEGER,
cont>                 y COMPUTED BY (SELECT COUNT(*) FROM
cont>                   t1 WHERE t1.col1 = t2.x));
```

```
SQL> --
SQL> -- Assume values have been inserted into the tables.
SQL> --
SQL> SELECT * FROM t1;
        COL1         COL2
           1          100
           1          101
           1          102
           2          200
           3          300
5 rows selected
SQL> SELECT * FROM t2;
           X            Y
           1            3
           3            1
2 rows selected
SQL> --
SQL> DROP TABLE t1 CASCADE;
Computed Column Y in table T2 is being set to NULL.
SQL> SELECT * FROM t2;
           X            Y
           1            NULL
           3            NULL
```

You can alter the table and delete the computed-by column. At some future point, you can then alter the table and create a new computed-by column using the same name but with a different computed-by expression.

However, a computed-by column is not set to NULL if it references a declared local temporary table that has been deleted by a DROP TABLE CASCADE statement. An exception is raised if you query the declared local temporary table in this situation.

- When a table is deleted using CASCADE, any computed-by columns in global or local temporary tables that reference the deleted table will be set to NULL. However, because temporary tables cannot be altered, the only way to use the temporary table (with the updated metadata) is to disconnect from the database and re-attach. For example:

```
SQL> CREATE TABLE test_table (x INT UNIQUE, y CHAR(4));
SQL> CREATE GLOBAL TEMP TABLE temp_tab (a INT,
cont>                b COMPUTED BY (SELECT y FROM test_table WHERE x = a),
cont>                c COMPUTED BY CHAR_LENGTH(b))
cont> ON COMMIT PRESERVE ROWS;
SQL> INSERT INTO test_table VALUES (1, 'xxxx');
SQL> INSERT INTO temp_tab (a) VALUES (1);
SQL> COMMIT;
SQL> DROP TABLE test_table CASCADE;
Constraint TEST_TABLE_UNIQUE_X is also being deleted.
Computed Column B in table TEMP_TAB is being set to NULL.
SQL> SELECT * FROM temp_tab;
%RDB-E-OBSOLETE_METADA, request references metadata objects that no
longer exist
-RDMS-F-BAD_SYM, unknown relation symbol - TEST_TABLE
SQL> COMMIT;
SQL> DISCONNECT ALL;
SQL> ATTACH 'FILENAME mf_personnel';
SQL> SELECT * FROM temp_tab;
0 rows selected
```

## Examples

Example 1: Deleting a table from an attached database

```
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> DROP TABLE PERS.DEGREES;
SQL> COMMIT;
```

Example 2: Deleting a table and definitions that reference it from the default database

```
SQL> ATTACH 'FILENAME corporate_data';
SQL> DROP TABLE ADMINISTRATION.PERSONNEL.EMPLOYEES CASCADE;
View ADMINISTRATION.PERSONNEL.REVIEW_DATE is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_INFO is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_SALARY is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_JOB is also being dropped.
Constraint ADMINISTRATION.RECRUITING.DEGREES_FOREIGN2 is also being dropped.
Constraint ADMINISTRATION.PERSONNEL.EMPLOYEES_PRIMARY_EMPLOYEE_ID is also
being dropped.
Constraint ADMINISTRATION.PERSONNEL.EMP_SEX_VALUES is also being dropped.
Constraint ADMINISTRATION.PERSONNEL.HOURLY_HISTORY_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.PERSONNEL.JOB_HISTORY_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.RECRUITING.RESUMES_FOREIGN2 is also being dropped.
Constraint ADMINISTRATION.PERSONNEL.SALARY_HISTORY_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.PERSONNEL.STATUS_CODE_VALUES is also being dropped.
Index ADMINISTRATION.PERSONNEL.EMP_LAST_NAME is also being dropped.
Index ADMINISTRATION.PERSONNEL.EMP_EMPLOYEE_ID is also being dropped.
Trigger ADMINISTRATION.PERSONNEL.EMPLOYEE_ID_CASCADE_DELETE is also being
dropped.
Trigger ADMINISTRATION.PERSONNEL.STATUS_CODE_CASCADE_UPDATE is also being
dropped.
```

## DROP TRIGGER Statement

Deletes a trigger definition from the physical database and, if the database was attached with PATHNAME, from the repository.

### Environment

You can use the DROP TRIGGER statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
DROP TRIGGER ───► <trigger-name> ───►
```

### Arguments

**trigger-name**
Specifies the name of the trigger to be deleted.

### Usage Notes

- Attempts to delete a trigger fail if that trigger is involved in a query at the same time. Users must detach from the database with a DISCONNECT statement before you can delete the trigger. When Oracle Rdb first accesses an object such as the table accessed by the trigger, a lock is placed on that object and not released until the user exits the database. If you attempt to update this object, you get a LOCK CONFLICT ON CLIENT message due to the other users' access to the object.

  Similarly, while you are deleting a trigger, users cannot execute queries involving tables referred to by the trigger until you have completed the transaction with a COMMIT or ROLLBACK statement for the DROP statement. The user receives a LOCK CONFLICT ON CLIENT error message. While DDL operations are performed, normal data locking mechanisms are used against system tables. (System tables contain information about objects in the database.) Therefore, attempts to update an object lock out attempts to query that object. These locks are held until the commit or rollback of the DDL operation.

The WAIT/NOWAIT clause of the SET TRANSACTION statement does not affect attempts to update metadata with simultaneous queries.

Even if you specify SET TRANSACTION WAIT for the metadata update transaction, you get the following error message if a lock conflict exists:

```
%RDB-E-LOCK_CONFLICT, request failed due to locked resource; no-wait
parameter specified for transaction
-RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-LCKCNFLCT, lock conflict on client
SQL>
```

However, a user's query waits for a metadata update to complete with a ROLLBACK or COMMIT statement, even if the user specified NOWAIT in the SET TRANSACTION statement.

- You must execute this statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- To delete a trigger, you must have DELETE access to the table for which the trigger is defined.
- Other users are allowed to be attached to the database when you issue the DROP TRIGGER statement.
- You cannot execute the DROP TRIGGER statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.

## Examples

Example 1: Deleting the EMPLOYEE_ID_CASCADE_DELETE trigger

```
SQL> ATTACH 'FILENAME personnel';
SQL> SHOW TRIGGERS
User triggers in database with filename PERSONNEL
     COLLEGE_CODE_CASCADE_UPDATE
     EMPLOYEE_ID_CASCADE_DELETE
     STATUS_CODE_CASCADE_UPDATE
SQL> DROP TRIGGER EMPLOYEE_ID_CASCADE_DELETE;
SQL> SHOW TRIGGERS
User trigggers in database with filename PERSONNEL
     COLLEGE_CODE_CASCADE_UPDATE
     STATUS_CODE_CASCADE_UPDATE
SQL>
```

## DROP VIEW Statement

Deletes the specified view definition. When the DROP VIEW statement executes, SQL deletes the view definition from the database. If you attach to the database using the PATHNAME qualifier, SQL also deletes the view definition from the repository.

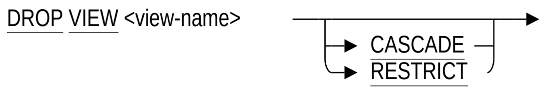
You can delete a view definition even when there are active users. Deleting a view definition does not affect active users until you commit your transaction, the users exit their sessions, and they declare the database again. SQL automatically deletes any views that refer to the view named in the DROP VIEW statement.

### Environment

You can use the DROP VIEW statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**view-name**
Specifies the name of the view definition you want to delete.

**CASCADE**
Specifies that you want SQL to delete all other view definitions that refer to the named view and then delete that view definition. This is known as a cascading delete. For modules, if you delete a view referenced by a stored routine with a routine or language-semantic dependency, SQL also marks the affected stored routines as invalid.

**RESTRICT**
Specifies that you want SQL to delete only the named view definition. If there are other views that refer to the named view, the deletion fails. For modules, because the DROP DOMAIN RESTRICT statement fails when you attempt to delete a view referenced in a stored routine, the dependent stored routine is not invalidated.

## Usage Notes

- If you do not specify either the CASCADE keyword or the RESTRICT keyword, SQL executes a RESTRICT delete by default:

  ```
  SQL> ATTACH 'FILENAME personnel';
  SQL> DROP VIEW CURRENT_INFO;
  ```

- You must execute the DROP VIEW statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL implicitly starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.
- The DROP VIEW statement fails when both of the following circumstances are true:
  - The database to which it applies was created with the DICTIONARY IS REQUIRED argument.
  - The database was attached using the FILENAME argument.

  Under these circumstances, the statement fails with the following error when you issue it:

  ```
  %RDB-E-NO_META_UPDATE, metadata update failed
  -RDMS-F-CDDISREQ, CDD required for metadata updates is not being maintained
  ```

- Because Oracle Rdb creates dependencies between stored routines and metadata (like views) on which they are compiled and stored, you can delete a view with a routine or language-semantic dependency if you specify CASCADE but you cannot with RESTRICT. In the case of DROP VIEW CASCADE, when the view referenced in a stored routine is deleted, the stored routine is marked as invalid in the RDB$INTERRELATIONS system table. In the case of DROP VIEW RESTRICT, because the statement fails when you attempt to delete the view referenced in a stored routine, the dependent stored routine is not invalidated. Refer to the CREATE MODULE Statement for a list of statements that can or cannot cause stored routine invalidation.

Refer to the *Oracle Rdb7 Guide to SQL Programming* for detailed information about stored routine dependency types and how metadata changes can cause invalidation of stored routines.

- If you are using Oracle Rdb, you cannot execute the DROP VIEW statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- If a deleted view is referenced in a computed-by column, the computed-by column is set to NULL. However, if the computed-by column is part of the definition of a declared local temporary table, an exception is raised.

## Examples

Example 1: Deleting a view definition

The following example deletes the view definition CURRENT_INFO:

```
SQL> DROP VIEW CURRENT_INFO;
SQL> COMMIT;
```

Example 2: Deleting a view with dependent views

This example shows that SQL will not automatically delete any views that refer to the view named in the DROP VIEW statement. You must use the CASCADE keyword to delete a view with dependent views.

```
SQL> DROP VIEW CURRENT_JOB;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-VIEWINVIEW, view CURRENT_JOB is referenced by view CURRENT_INFO
-RDMS-F-VIEWNOTDEL, view CURRENT_JOB has not been deleted

SQL> DROP VIEW CURRENT_JOB CASCADE;
View CURRENT_INFO is also being dropped.
SQL> COMMIT;
```

## EDIT Statement

Calls an editor that lets you modify the SQL statements you issued within a terminal session.

SQL supports a variety editors, some of which are:

- EDT, which is the default SQL editor on OpenVMS
- DEC Text Processing Utility (DECTPU) editors on OpenVMS, such as EVE
- VAX Language-Sensitive Editor (LSE) on OpenVMS, which is based on DECTPU and provides templates that guide you in entering syntactically correct statements
- vi (visual editor), a screen editor which is the default SQL editor on Digital UNIX

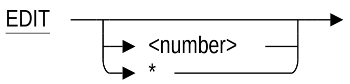
To invoke an editor other than the default, you must define the SQL$EDIT logical name or SQL_EDIT configuration parameter. See the Usage Notes section for details.

You can use the editor you choose with your usual initialization file to modify your previous SQL statements, construct your next statement or group of statements, or include a file with other statements.

### Environment

You can issue the EDIT statement only in interactive SQL.

### Format



### Arguments

**number**
Specifies the number of previous statements you want to edit, up to the number specified in the last SET EDIT KEEP statement. If you specify zero as the number, then SQL does not include any statements in the editing buffer. If you omit the number argument, SQL includes the last statement issued in the editing buffer.

*** (asterisk)**
Specifies a wildcard character. If you use the * (asterisk) wildcard character, SQL includes in the editing buffer the number of statements specified in the last SET EDIT KEEP statement. If you do not use the SET EDIT KEEP statement, EDIT * puts the last 20 statements in your editing buffer. If you omit the * (asterisk) wildcard character, SQL includes the last statement issued in the editing buffer.

## Usage Notes

- To invoke an editor other than the vi default for Digital UNIX, you can do the following:
  - Define the SQL_EDIT configuration parameter in your .dbsrc file.
  - Define the EDITOR environment variable.

  The following example shows how to define the SQL_EDIT configuration parameter in your .dbsrc file to specify the Emacs editor:

  ```
  SQL_EDIT emacs
  ```
- When you use the EDIT statement, the following sequence occurs:
  1. SQL invokes the editor specified by the SQL$EDIT logical name, the SQL_EDIT configuration parameter, or the EDITOR environment variable and initializes the editor according to your initialization file for that editor, if any. If you do not have an initialization file, SQL uses the system default editor.
  2. SQL places the statements you asked for in the editing buffer.

     If you are using an editor other than EDT, DECTPU, or LSE, SQL places the statements in a temporary file and spawns a subprocess to execute the command you specified in the SQL$EDIT logical name, SQL_EDIT configuration parameter, or the EDITOR environment variable.
  3. The SQL prompt (SQL>) disappears and is replaced by the normal display for the editor.
  4. You can now edit the SQL statements.

     If you are using the EDT, DECTPU, or LSE editor, SQL automatically executes all the statements in the main editing buffer when you exit from the editor. If you are using an editor other than EDT, DECTPU, or LSE, you are prompted whether or not you want to execute the command lines in the main editing buffer when you exit the editor. A later Usage Note explains how to bypass this prompt and execute the command lines automatically with other editors.

If you quit from the editor, SQL returns to the command level and displays the SQL prompt (SQL>) without executing a statement.

OpenVMS VAX OpenVMS Alpha

- You do not need to do anything to specify EDT as the editor to use within interactive SQL because it is the OpenVMS system default editor. To use DECTPU, it must be installed on your system, and you must define the logical name SQL$EDIT. To use LSE, it must be installed on your system, and you must define the logical names SQL$EDIT and LSE$ENVIRONMENT.

```
$ ! To specify DECTPU as your editor in interactive SQL:
$ DEFINE SQL$EDIT TPU
$ !
$ ! To specify LSE as your editor in interactive SQL:
$ DEFINE SQL$EDIT LSE
$ DEFINE LSE$ENVIRONMENT -
_$ SYS$COMMON:[SYSLIB]LSE$SYSTEM_ENVIRONMENT.ENV
```

  Then, when you type EDIT in an SQL session, SQL calls the editor specified by the SQL$EDIT logical name. If SQL$EDIT is not defined or is defined to be something other than DECTPU or LSE, then SQL invokes the EDT editor when you issue the EDIT command. If SQL cannot find the DECTPU or LSE shareable image, it invokes EDT.

- SQL supports two sets of LSE templates; one for interactive SQL and one for SQL module language.

  **Note**

  The LSE templates provided with Oracle Rdb only provide support for SQL syntax through V4.2. The templates do not provide support for new and changed syntax after V4.2.

  The templates guide you in entering syntactically correct statements. The templates for interactive SQL can be invoked within interactive SQL (as explained earlier in these Usage Notes) or at the DCL level. The templates for SQL module language can be invoked only at the DCL level.

  To invoke the SQL templates at the DCL level, type LSE followed by the name of the file you want to edit. The file extension determines the set of templates that LSE uses. For interactive SQL, use the .SQL file extension; for the SQL module language, use .SQLMOD. For example, you can type the following commands:

```
$ ! To invoke the LSE templates for interactive SQL at DCL level:
$ LSE SAMPLE.SQL
$ ! To invoke the LSE templates for module language:
$ LSE SAMPLE.SQLMOD
```

- If you specify an editor based on DECTPU for use in interactive SQL (through the SQL$EDIT logical), you cannot always read or write files from the editing buffer created when you issue the interactive SQL statement EDIT.
  - In EVE editors, the INCLUDE command to read a file into the default editing buffer fails. To work around this problem, you must use the GET FILE command to place the file in another buffer and copy the buffer to the MAIN buffer that SQL executes upon exiting from the editor.
  - In all editors based on DECTPU, the DECTPU WRITE_FILE command (WRITE in EVE) to write the default editing buffer fails. You must copy the default buffer to another buffer and write that buffer to a file. ♦
- If you execute an SQL statement and then execute the HELP statement to read the help text, an EDIT statement puts only the original SQL statement in the editing buffer, not the HELP statement. For example:

```
SQL> SHOW TABLE (COLUMN) CURRENT_JOB;
Information for table CURRENT_JOB

Columns for view CURRENT_JOB:
Column Name                     Data Type        Domain
-----------                     ---------        ------
LAST_NAME                       CHAR(14)
FIRST_NAME                      CHAR(10)
EMPLOYEE_ID                     CHAR(5)
JOB_CODE                        CHAR(4)
DEPARTMENT_CODE                 CHAR(4)
SUPERVISOR_ID                   CHAR(5)
JOB_START                       DATE VMS
SQL>
SQL> HELP

  Information available:

  $          @          alias      ALTER      arith_expression      ATTACH
  authorization_id      BEGIN_DECLARE         CLOSE      col_select_expr
  Command_recall        COMMENT_ON COMMIT     CONNECT    correlation_name
   .
   .
   .
```

```
Topic? SHOW TABLES
   .
   .
   .
SQL> EDIT
    600000       SHOW TABLE (COLUMN) CURRENT_JOB;
*<CHANGE>

  SHOW TABLE (COLUMN) CURRENT_JOB;
*<QUIT>
```

OpenVMS VAX OpenVMS Alpha

- Interactive SQL users can recall the 20 most recent command lines using the up and down arrow keys or the Ctrl/B key sequence.
  - The up arrow key recalls lines in sequence from most recent to least recent.
  - The Ctrl/B key sequence also recalls lines in sequence from most recent to least recent.
  - After you recalled prior lines, the down arrow key allows you to recall more recently entered lines. ♦
- The EDIT statement does not allow any operating system invocation statements or executable statements in the buffer of statements to edit.

  If you enter one of these statements in the edit buffer, it generates an error when you exit from the editor. For example, on OpenVMS:

```
SQL> SHOW VIEW;
User tables in database with filename personnel
     CURRENT_INFO                   A view.
     CURRENT_JOB                    A view.
     CURRENT_SALARY                 A view.
SQL> --
SQL> -- Make a mistake with DCL DIRECTORY statement:
SQL> $ DIP
%DCL-W-IVVERB, unrecognized command verb - check validity and spelling
 \DIP\
SQL> EDIT
   4600000       SHOW VIEWS;
*<CHANGE>

SHOW VIEWS;
```

```
*<QUIT>
SQL> -- Type EDIT again and add the $ DIR command.
SQL> EDIT
   4600000       SHOW VIEWS;
*<CHANGE>

SHOW VIEWS;
$ DIR

*<EXIT>
  2 lines
```

```
SHOW VIEWS;
User tables in database with filename personnel
     CURRENT_INFO                    A view.
     CURRENT_JOB                     A view.
     CURRENT_SALARY                  A view.
$ DIR
$ DIR
^
%SQL-F-LOOK_FOR_STMT, Syntax error, looking for a valid SQL statement,
 found $ instead
```

The statement $ DIP does not appear in the editing buffer because it is interpreted as a DCL statement. If you type $ DIR while in the editor and then exit, SQL generates a syntax error as shown in the previous example.

- If you are using an editor other than EDT, DECTPU, or LSE, you are prompted whether or not you want to execute the command lines in the main editing buffer when you exit the editor. You can bypass this prompt by setting the SQL$EDIT_TWO logical name or SQL_EDIT_TWO configuration parameter.

  The SQL$EDIT_TWO logical name or SQL_EDIT_TWO configuration parameter can be set to true so that the editor accepts an input file followed by an output file. The editor edits the output file and inserts the contents of the input file. Writing out the output file signals SQL to execute the command lines. In order for the SQL$EDIT_TWO logical name or SQL_EDIT_TWO configuration parameter to be useful, the SQL$EDIT logical name or SQL_EDIT configuration parameter must also be set.

  For example, on Digital UNIX, one way to make use of the SQL_EDIT_TWO configuration parameter is to write a one-line script to invoke the editor to process the input and output file specifications in the appropriate manner. For example:

```
$ cat sqledit
emacs -nw $2 -insert $1
$
```

Then, using the previous example, setting SQL_EDIT to sqledit and SQL_ EDIT_TWO to true invokes Emacs and inserts the contents of the input file specification into the output file specification. Then upon exiting the editor, the command lines in the main editing buffer are executed automatically.

If the SQL$EDIT_TWO logical name or the SQL_EDIT_TWO configuration parameter is not set to true, then the editor is invoked with only one file specification and, upon exiting, you are prompted whether or not you want to execute the command lines in the main editing buffer.

## Examples

Example 1: Correcting a misspelled statement

1. Make a mistake:

```
SQL> SELECT JOB_TITLE FROM JOSB;
%SQL-F-RELNOTDEF, Table JOSB is not defined in schema
SQL>
```

2. Invoke the editor:

```
SQL> EDIT
```

3. When in the editor, change JOSB to JOBS. See the manual for the editor you are using for detailed editing instructions.
4. Exit from the editor. SQL automatically executes the contents of the editing buffer.

```
* EXIT
SELECT JOB_TITLE FROM JOBS;
 Associate Programmer
 Clerk
 Assistant Clerk
 Department Manager
 Dept. Supervisor
       .
       .
       .
```

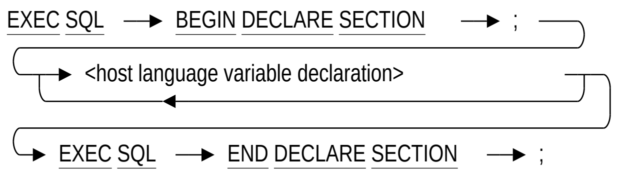
## END DECLARE Statement

Delimits the end of a host language variable declaration section in a precompiled program.

### Environment

You can use the END DECLARE statement embedded in host language programs to be precompiled.

### Format



### Arguments

**BEGIN DECLARE SECTION**
Delimits the beginning of a host language variable declaration.

**; (semicolon)**
Terminates the BEGIN DECLARE and END DECLARE statements.

Which terminator you use depends on the language in which you are embedding the host language variable. The following table shows which terminator to use:

| | Required SQL Terminator | |
|---|---|---|
| **Host Language** | **BEGIN DECLARE Statement** | **END DECLARE Statement** |
| COBOL | END-EXEC | END-EXEC |
| FORTRAN | None required | None required |
| Ada, C, Pascal, or PL/I | ; (semicolon) | ; (semicolon) |

**host language variable declaration**
Specifies a variable declaration embedded within a program.

See Section 2.2.19 for more information on host language variable definitions.

**END DECLARE SECTION**
Delimits the end of host language variable declarations.

## Usage Notes

- The ANSI/ISO SQL standard specifies that host language variables used in embedded SQL statements must be declared within a pair of embedded SQL BEGIN DECLARE . . . END DECLARE statements. If ANSI/ISO compliance is important for your application, you should include all declarations for host language variables used in embedded SQL statements within a BEGIN DECLARE . . . END DECLARE block.
- SQL does not require that you enclose host language variables with BEGIN DECLARE and END DECLARE statements. SQL does, however, issue a warning message if both of the following conditions exist:
  - Your program includes a section delimited by BEGIN DECLARE and END DECLARE statements.
  - You refer to a host language variable that is declared outside the BEGIN DECLARE and END DECLARE section.
- In addition to host language variable declarations, you can include other host language statements in a BEGIN DECLARE . . . END DECLARE section. See Section 2.2.19 and the BEGIN DECLARE Statement for more details.

## Examples

OpenVMS VAX OpenVMS Alpha

Example 1: Declaring a host language variable within a BEGIN . . . END DECLARE block

The following example shows portions of a PL/I program. The first part of the example declares the host language variable LNAME within the BEGIN DECLARE and END DECLARE statements. The semicolon is necessary as a terminator because the language is PL/I.

The second part of the example shows a singleton SELECT statement that specifies a one-row result table. The statement assigns the value in the row to the previously declared host language variable LNAME.

```
EXEC SQL
BEGIN DECLARE SECTION;
  DECLARE LNAME char(20);
EXEC SQL
END DECLARE SECTION;
.
.
.
EXEC SQL
SELECT FIRST_NAME
   INTO :LNAME
   FROM EMPLOYEES
   WHERE EMPLOYEE_ID = "00164";
```

♦

## Execute ( @ ) Statement

In SQL, the at sign ( @ ) means execute. When you type @ and the name of an indirect command file, SQL executes the statements in that file as if you typed them one-at-a-time at the SQL prompt (SQL>). The command file must be a text file that contains SQL statements.

The default file extension for an indirect command file is .SQL.

You can use the SET VERIFY statement to display the commands in the file as they execute.

SQL recognizes a special SQL command file called SQLINI.SQL, which contains SQL statements to be issued before SQL displays the SQL prompt (SQL>). If this file exists, SQL executes the commands in the file first, before displaying the prompt and accepting your input. If you define the logical name or configuration parameter SQLINI to point to a general initialization file, SQL uses this file. Otherwise, it looks for SQLINI.SQL in the current default directory.

### Environment

You can issue the execute ( @ ) statement only in interactive SQL.

### Format

```
@ <file-spec>
```

### Arguments

**file-spec**
Specifies the name of an indirect command file. You can use either a full file specification, a file name, or a logical name on OpenVMS or a link on Digital UNIX. If you use a file name, SQL looks in the current default directory for a file by that name. The file must contain valid SQL statements.

## Usage Notes

Interactive SQL interprets any command line that begins with an at sign (@) as the start of a command file invocation. This is true even if the at sign is a continuation of a string literal from the previous line, which can lead to confusing results.

```
SQL> INSERT INTO EMPLOYEES (CITY) VALUES ('AtSign -
cont> @City')
%SQL-F-FILEACCERR, Error parsing name of file City')
-RMS-F-SYN, file specification syntax error
SQL> --
SQL> -- You can avoid errors by breaking your statement line elsewhere:
SQL> --
SQL> INSERT INTO EMPLOYEES (CITY) VALUES
cont> ('AtSign - @City');
1 row inserted
```

## Examples

Example 1: Storing interactive SQL statements in a startup file

You can use an indirect command file to specify characteristics of your SQL terminal session. This example assumes that SQLINI is defined as a logical name or configuration parameter that points to the file setup.sql. The file contains the following SQL statements:

```
SET VERIFY;
SET EDIT KEEP 5; -- This line will be displayed on the terminal
```

SQL executes the file when you invoke interactive SQL.

```
$ SQL
SQL> SET EDIT KEEP 5; -- This line will be displayed on the terminal
SQL>
```

When it executes, setup.sql turns on the indirect command file display and limits the number of statements saved by SQL for editing to five.

Example 2: Executing frequently used queries

The file EMPADDR.SQL contains the following SQL statements:

```
-- This command file generates information for a mailing list.
--
ATTACH 'FILENAME personnel';
SET OUTPUT MAILLIST.DOC
SELECT  FIRST_NAME, MIDDLE_INITIAL, LAST_NAME,
        ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE
FROM EMPLOYEES;
--
-- Execute the file by using the following command:
--
@EMPADDR
```

Example 3: Using a logical name or link to run a command file

If you define COUNT to be a logical name or link to a file, you can use the command @COUNT to execute the statements in the file, even if the file is located in a directory other than the default directory. The file COUNT.SQL contains the following SQL statements:

```
-- This command file counts the rows in
--  each table of the personnel database.
--
SET NOVERIFY;
SELECT 'Count of Employees -------> ', COUNT (*) FROM EMPLOYEES;
SELECT 'Count of Jobs -----------> ', COUNT (*) FROM JOBS;
SELECT 'Count of Degrees ---------> ', COUNT (*) FROM DEGREES;
SELECT 'Count of Salary_History --> ', COUNT (*) FROM SALARY_HISTORY;
SELECT 'Count of Job_History -----> ', COUNT (*) FROM JOB_HISTORY;
SELECT 'Count of Work_Status -----> ', COUNT (*) FROM WORK_STATUS;
SELECT 'Count of Departments -----> ', COUNT (*) FROM DEPARTMENTS;
SELECT 'Count of Colleges --------> ', COUNT (*) FROM COLLEGES;
```

The following example shows how to execute the file and the output:

```
$ SQL
SQL> @COUNT;

 Count of Employees ------->            100
1 row selected

 Count of Jobs ------------>             15
1 row selected

 Count of Degrees --------->            166
1 row selected
   .
   .
   .
```

## EXECUTE Statement

Dynamically executes a previously prepared statement other than SELECT.

The EXECUTE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not necessarily part of a program's source code, but can be generated while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

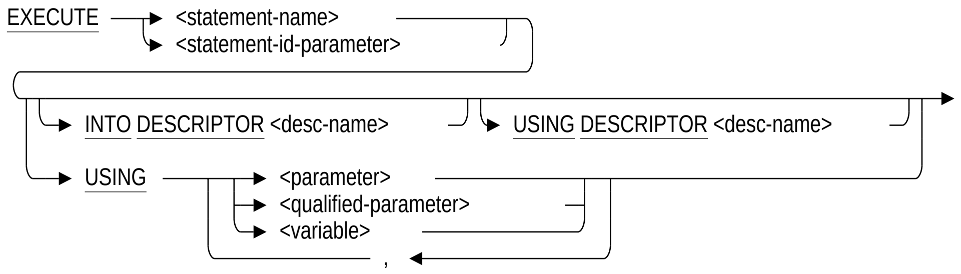
If a program needs to dynamically execute a statement more than once, the statement should be prepared first with the PREPARE statement and executed each time with the EXECUTE statement. SQL does not parse and compile prepared statements every time it dynamically executes them with the EXECUTE statement.

### Environment

You can use the EXECUTE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

### Format



## Arguments

**statement-name**
**statement-id-parameter**
Specifies the name of a prepared statement other than a prepared SELECT statement. (Prepared SELECT statements are executed through embedded DECLARE CURSOR, OPEN, and FETCH statements.) You can supply either a parameter or a compile-time statement name. Specifying a parameter lets SQL supply identifiers to programs at run time. Use an integer parameter to contain the statement identifier returned by SQL or a character string parameter to contain the name of the statement that you pass to SQL.

If the PREPARE statement for the dynamically executed statement specifies a parameter, use that same parameter in the EXECUTE statement instead of an explicit statement name.

**INTO DESCRIPTOR descriptor-name**
Specifies an SQLDA descriptor that contains addresses and data types that specify output parameters or variables.

The descriptor must be a structure declared in the host language program as an SQLDA. If the program is precompiled and uses the embedded SQL statement INCLUDE SQLDA, the name of the structure is simply SQLDA. Programs can use multiple SQLDAs, but must explicitly declare them with names other than SQLDA.

Programs can always use the INTO DESCRIPTOR clause of the EXECUTE statement whether or not the statement string contains output parameter markers, as long as the value of the SQLD field in the SQLDA corresponds to the number of output parameter markers. SQL updates the SQLD field with the correct number of output parameter markers when it processes the DESCRIBE statement for the statement string.

**USING DESCRIPTOR descriptor-name**
Specifies an SQLDA descriptor that contains addresses and data types of output parameters or variables.

The descriptor must be a structure declared in the host language program as an SQLDA. If the program is precompiled and uses the embedded SQL statement INCLUDE SQLDA, the name of the structure is simply SQLDA. Programs can use multiple SQLDAs, but must explicitly declare them with names other than SQLDA.

Programs can always use the USING DESCRIPTOR clause of the EXECUTE statement whether or not the statement string contains input parameter markers, as long as the value of the SQLD field in the SQLDA corresponds to the number of input parameter markers. SQL updates the SQLD field with the correct number of input parameter markers when it processes the DESCRIBE statement for the statement string.

**USING parameter**
**USING qualified-parameter**
**USING variable**
Specifies input parameters or variables whose values SQL uses to replace parameter markers in the prepared statement string.

When you specify a list of parameters or variables, the number of parameters in the list must be the same as the number of input parameter markers in the statement string of the prepared statement. If the program determines that a statement string had no input parameter markers, the USING clause is not allowed.

## Usage Notes

- You can specify the INTO DESCRIPTOR clause in the EXECUTE statement without additional clauses. However, if you specify the USING clause with the INTO DESCRIPTOR clause, you can only specify the keyword DESCRIPTOR. Parameters and variables are not allowed. See the following example for the correct syntax:

  ```
  EXECUTE statement-name
  INTO DESCRIPTOR desc-name USING DESCRIPTOR desc-name;
  ```

- When you issue the EXECUTE statement for a previously prepared statement, you might want to obtain information beyond the success or failure code returned in the SQLCODE status parameter. For example, you might want to know how many rows were affected by the execution of an INSERT, DELETE, UPDATE, FETCH, or SELECT statement. SQL returns this information in the SQLERRD[2] field of the SQLCA.

  However, if you use an SQLCA parameter when you execute a prepared statement, you must use an SQLCA parameter when you prepare that statement. For example, using SQL module language calls from C, your code might look like the following where the SQLCA parameter is passed to both procedures:

```
static struct SQLCA  sqlca;
/* ... */
PREPARE_STMT(&sqlca, statement, &stmt_id);
/* ... */
EXECUTE_STMT(&sqlca, &stmt_id);
```

For more information about the SQLCA, including the SQLERRD[2] field, see Appendix B.

## Example

Example 1: Executing an INSERT statement with parameter markers

These fragments from the online sample C program sql_dynamic illustrate using an EXECUTE statement in an SQL module procedure to execute a dynamically generated SQL statement.

The program accepts input of any valid SQL statement from the terminal and calls the subunit shown in the following program excerpt:

```
   .
   .
   .
/*
**-----------------------------------------------------------------------------
**  Begin Main routine
**-----------------------------------------------------------------------------
*/

  int   sql_dynamic (psql_stmt, input_sqlda, output_sqlda, stmt_id, is_select)
  char  *psql_stmt;
  sqlda *input_sqlda;
  sqlda *output_sqlda;
  long  *stmt_id;
  int   *is_select;

{
    sqlda sqlda_in, sqlda_out;    /* Declare the SQLDA structures. */
    int rowcount, status;
    int param;

/* Declare arrays for storage of original data types and allocate memory. */

    mem_ptr output_save;
    mem_ptr input_save;

    /* * If a NULL SQLDA is passed, then a new statement is being prepared. */

    if ((*input_sqlda == NULL) && (*output_sqlda == NULL))
         {
         new_statement = TRUE;
```

```
 /*
 * Allocate separate SQLDAs for parameter markers (SQLDA_IN) and select
 * list items (SQLDA_OUT).  Assign the value of the constant MAXPARMS
 * to the SQLN field of both SQLDA structures.  SQLN specifies to
 * SQL the maximum size of the SQLDA.
 */

if ((sqlda_in = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
    {
    printf ("\n\n*** Error allocating memory for sqlda_in: Abort");
    return (-1);
    }
else    /* set # of possible parameters */
    sqlda_in->sqln = MAXPARAMS;

if ((sqlda_out = (sqlda) calloc (1, sizeof (sqlda_rec))) == 0)
    {
    printf ("\n\n*** Error allocating memory for sqlda_out: Abort");
    return (-1);
    }

    }
else
    /* Set # of possible select list items. */
    sqlda_out->sqln = MAXPARAMS;

/* copy name SQLDA2 to identify the SQLDA */

strncpy(&sqlda_in->sqldaid[0],"SQLDA2  ",8);
strncpy(&sqlda_out->sqldaid[0],"SQLDA2  ",8);

/*
* Call an SQL module language procedure, prepare_stmt and
* describe_stmt that contains a PREPARE and DESCRIBE...SELECT_LIST
* statement to prepare the dynamic statement and write information
* about any select list items in it to SQLDA_OUT.
*/

*stmt_id = 0;  /* If <> 0 the BADPREPARE error results in the PREPARE.*/

PREPARE_STMT (&SQLCA, stmt_id, psql_stmt);
if (SQLCA.SQLCODE  != sql_success)
    {
    printf ("\n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
            SQLCA.SQLCODE);
    display_error_message();
    return (-1);
    }
    DESCRIBE_SELECT (&SQLCA, stmt_id, sqlda_out);
    if (SQLCA.SQLCODE  != sql_success)
        {
        printf ("\n\nDSQL-E-PREPARE, Error %d encountered in PREPARE",
                    SQLCA.SQLCODE);
        display_error_message();
        return (-1);
        }

    /*
    * Call an SQL module language procedure, describe_parm, that contains a
    * DESCRIBE...MARKERS statement to write information about any parameter
    * markers in the dynamic statement to sqlda_in.
    */

    DESCRIBE_PARM (&SQLCA, stmt_id, sqlda_in);
    if (SQLCA.SQLCODE  != sql_success)
        {
        printf ("\n\n*** Error %d returned from describe_parm: Abort",
                 SQLCA.SQLCODE);
        display_error_message();
        return (-1);
        }

    /* Save the value of the SQLCA.SQLERRD[1] field so that program can
     *  determine if the statement is a SELECT statement or not.
     *  If the value is 1, the statement is a SELECT statement.*/

       *is_select = SQLCA.SQLERRD[1];
        .
        .
        .
/*
* Check to see if the prepared dynamic statement contains any parameter
* markers by looking at the SQLD field of sqlda_in.  SQLD contains the
* number of parameter markers in the prepared statement. If SQLD is
* positive, the prepared statement contains parameter markers.  The program
* executes a local procedure, get_in_params, that prompts the user for
* values, allocates storage for those values, and updates the SQLDATA field
* of sqlda_in:
*/

if (sqlda_in->sqld > 0)
    if ((status = get_in_params(sqlda_in,input_save)) != 0)
        {
        printf ("\nError returned from GET_IN_PARAMS. Abort");
        return (-1);
        }
    /* Check to see if the prepared dynamic statement is a SELECT by looking
    * at the value in is_select, which stores the value of the
    * SQLCA.SQLERRD[1] field.  If that value is equal to 1, the prepared
    * statement is a SELECT statement.  The program allocates storage for
    * rows for SQL module language procedures to open and fetch from a cursor,
    * and displays the rows on the terminal:
    */

    if (*is_select)
        {
        if (new_statement == TRUE)      /* Allocate buffers for output. */
            {
            /* assign a unique name for the cursor */
            sprintf(cursor_name,"%2d",++cursor_counter);

            if ((status = allocate_buffers(sqlda_out)) != 0)
     .
     .
     .
/*
* If the SQLCA.SQLERRD[1] field is not 1, then the prepared statement is not a
* SELECT statement and only needs to be executed.  Call an SQL module language
* procedure to execute the statement, using information about parameter
* markers stored in sqlda_in by the local procedure get_in_params:
*/
        {
        EXECUTE_STMT (&SQLCA, stmt_id, sqlda_in);
        if (SQLCA.SQLCODE != sql_success)
   .
   .
   .
```

The SQL module language procedures called by the preceding fragment:

```
   .
   .
   .
-------------------------------------------------------------------------------
-- Procedure Section
-------------------------------------------------------------------------------

-- This procedure prepares a statement for dynamic execution from the string
-- passed to it.  It also writes information about the number and data type of
-- any select list items in the statement to an SQLDA2 (specifically,
-- the sqlda_out SQLDA2 passed to the procedure by the calling program).
--

PROCEDURE PREPARE_STMT
    SQLCA
    :DYN_STMT_ID      INTEGER
    :STMT            CHAR(1024);

    PREPARE :DYN_STMT_ID FROM :STMT;
```

```
-- This procedure writes information to an SQLDA (specifically,
-- the sqlda_in SQLDA passed to the procedure by the calling program)
-- about the number and data type of any parameter markers in the
-- prepared dynamic statement.  Note that SELECT statements may also
-- have parameter markers.

PROCEDURE DESCRIBE_SELECT
    SQLCA
    :DYN_STMT_ID    INTEGER
    SQLDA;

    DESCRIBE :DYN_STMT_ID SELECT LIST INTO SQLDA;

PROCEDURE DESCRIBE_PARM
    SQLCA
    :DYN_STMT_ID INTEGER
    SQLDA;

    DESCRIBE :DYN_STMT_ID MARKERS INTO SQLDA;


-- This procedure dynamically executes a non-SELECT statement.
-- SELECT statements are processed by DECLARE CURSOR, OPEN CURSOR,
-- and FETCH statements.
--
-- The EXECUTE statement specifies an SQLDA2 (specifically,
-- the sqlda_in SQLDA2 passed to the procedure by the calling program)
-- as the source of addresses for any parameter markers in the dynamic
-- statement.
--
-- The EXECUTE statement with the USING DESCRIPTOR clause
-- also handles statement strings that contain no parameter markers.
-- If a statement string contains no parameter markers, SQL sets
-- the SQLD field of the SQLDA2 to zero.

PROCEDURE EXECUTE_STMT
    SQLCA
    :DYN_STMT_ID INTEGER
    SQLDA;

    EXECUTE :DYN_STMT_ID USING DESCRIPTOR SQLDA;
   .
   .
   .
```

## EXECUTE IMMEDIATE Statement

Dynamically prepares, executes, and releases an SQL statement.

The EXECUTE IMMEDIATE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to precompiled statements, which must be embedded in the program before it is compiled. Unlike embedded statements, such dynamically executed SQL statements are not necessarily part of the program's source code, but can be created while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

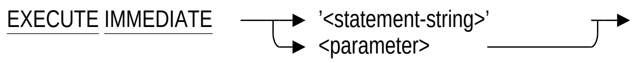
In the EXECUTE IMMEDIATE statement, the SQL statement cannot be a SELECT statement or contain parameter markers. However, if the statement meets those restrictions and will be dynamically executed only once, use the EXECUTE IMMEDIATE statement instead of PREPARE and EXECUTE statements.

### Environment

You can use the EXECUTE IMMEDIATE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

### Format



### Arguments

**statement-string**
**parameter**
Specifies the SQL statement to be prepared and executed dynamically. You either specify the statement string directly in a character string literal enclosed in single quotation marks, or in a parameter that contains the statement string.

Whether specified directly or by a parameter, the statement string must be a character string that is a dynamically executable SQL statement other than the SELECT statement (the PREPARE Statement lists the SQL statements that can be dynamically executed). The form for the statement is the same as in embedded SQL, except that you do not need to begin the string with EXEC SQL or end it with any statement terminator.

## Example

Example 1: Executing an INSERT statement with the EXECUTE IMMEDIATE statement

This COBOL program illustrates using the EXECUTE IMMEDIATE statement to prepare and execute a dynamic INSERT statement. Compare this example with the example for the EXECUTE statement (see the EXECUTE Statement), which uses an INSERT statement with parameter markers and displays the result of the insert operation.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   EXECUTE_IMMEDIATE_EXAMPLE.
*
* Illustrate EXECUTE_IMMEDIATE with a dynamic INSERT statement.
*
DATA DIVISION.

WORKING-STORAGE SECTION.

* Variable for DECLARE SCHEMA:
01 FILESPEC     PIC X(20).

* Variables to hold values for
* storage in EMPLOYEES:
01 EMP_ID       PIC X(5).
01 FNAME        PIC X(10).
01 MID_INIT     PIC X(1).
01 LNAME        PIC X(14).
01 ADDR_1       PIC X(25).
01 ADDR_2       PIC X(25).
01 CITY         PIC X(20).
01 STATE        PIC X(2).
01 P_CODE       PIC X(5).
01 SEX          PIC X(1).
01 BDATE        PIC S9(11)V9(7) COMP.
01 S_CODE       PIC X(1).
```

```
* Indicator variables for retrieving
* the entire row, including columns we
* do not assign values to, from
* the EMPLOYEES table:
01 EMP_ID_IND   PIC S9(4) COMP.
01 FNAME_IND    PIC S9(4) COMP.
01 MID_INIT_IND PIC S9(4) COMP.
01 LNAME_IND    PIC S9(4) COMP.
01 ADDR_1_IND   PIC S9(4) COMP.
01 ADDR_2_IND   PIC S9(4) COMP.
01 CITY_IND     PIC S9(4) COMP.
01 STATE_IND    PIC S9(4) COMP.
01 P_CODE_IND   PIC S9(4) COMP.
01 SEX_IND      PIC S9(4) COMP.
01 BDATE_IND    PIC S9(4) COMP.
01 S_CODE_IND   PIC S9(4) COMP.

* Buffer for error handling:
01 BUFFER       PIC X(300).
01 LEN          PIC S9(4) USAGE IS COMP.

* 01 disp_sqlcode       pic s9(9) sign leading separate.

* Load definition for SQL Communication Area (SQLCA):
EXEC SQL        INCLUDE SQLCA END-EXEC.

********************************************************************
*
*               P R O C E D U R E   D I V I S I O N
*
********************************************************************
PROCEDURE DIVISION.
START-UP.

* Assign value to FILESPEC:
        MOVE "SQL$DATABASE" TO FILESPEC

* Declare the schema:
        EXEC SQL DECLARE SCHEMA RUNTIME FILENAME :FILESPEC
        END-EXEC

*               Use an EXECUTE IMMEDIATE statement
*               to execute an INSERT statement:
            EXEC SQL EXECUTE IMMEDIATE
            "INSERT INTO EMPLOYEES
-               "(EMPLOYEE_ID,FIRST_NAME,LAST_NAME,CITY)
-               "VALUES ('99999','Les','Warton','Hudson')"
            END-EXEC
            PERFORM CHECK.

        PERFORM FETCHES.

EXEC SQL EXECUTE IMMEDIATE 'ROLLBACK' END-EXEC.
        PERFORM CHECK.

        DISPLAY "Rolled back changes.  All done.".
```

```
CLEAR-IT-EXIT.
        EXIT PROGRAM.

FETCHES.
        DISPLAY "Here’s the row we stored:"

        EXEC SQL PREPARE STMT FROM
        ’SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID = "99999"’
        END-EXEC
        EXEC SQL DECLARE C CURSOR FOR STMT END-EXEC

        EXEC SQL OPEN C END-EXEC
*       Clear values in host language
*       variables in case new values
*       from the table are null:
        MOVE SPACES TO EMP_ID
        MOVE SPACES TO FNAME
        MOVE SPACES TO MID_INIT
        MOVE SPACES TO LNAME
        MOVE SPACES TO ADDR_1
        MOVE SPACES TO ADDR_2
        MOVE SPACES TO CITY
        MOVE SPACES TO STATE
        MOVE SPACES TO P_CODE
        MOVE SPACES TO SEX
        MOVE ZERO TO BDATE
        MOVE SPACES TO S_CODE

        EXEC SQL FETCH C INTO
                :EMP_ID:EMP_ID_IND,
                :LNAME:LNAME_IND,
                :FNAME:FNAME_IND,
                :MID_INIT:MID_INIT_IND,
                :ADDR_1:ADDR_1_IND,
                :ADDR_2:ADDR_2_IND,
                :CITY:CITY_IND,
                :STATE:STATE_IND,
                :P_CODE:P_CODE_IND,
                :SEX:SEX_IND,
                :BDATE:BDATE_IND,
                :S_CODE:S_CODE_IND
        END-EXEC
        DISPLAY EMP_ID," ",
                FNAME," ",
                MID_INIT," ",
                LNAME," ",
                ADDR_1," ",
                ADDR_2," ",
                CITY," ",
                STATE," ",
                P_CODE," ",
                SEX," ",
                BDATE," ",
                S_CODE.

        PERFORM CHECK.
        EXEC SQL CLOSE C END-EXEC.

CHECK.
        IF SQLCODE NOT = 100 AND SQLCODE NOT = 0
                DISPLAY "Error: SQLCODE = ", SQLCODE
            CALL "SQL$GET_ERROR_TEXT" USING
                BY DESCRIPTOR BUFFER,
                BY REFERENCE LEN
            DISPLAY BUFFER(1:LEN)
        END-IF.
```

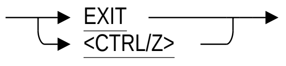
## EXIT Statement

Stops an interactive SQL session and returns you to the operating system prompt. By default, the EXIT statement commits changes made during the session.

### Environment

You can issue the EXIT statement in interactive SQL only.

### Format



### Usage Notes

- Both the QUIT and EXIT statements end an interactive SQL session. The QUIT statement automatically rolls back changes made during the session; the EXIT statement, by default, commits changes made during the session.
- If you have made uncommitted changes to the database when you issue the EXIT statement, SQL asks if you want to roll back the transaction.

  ```
  There are uncommitted changes to this database.
  Would you like a chance to ROLLBACK these changes (No)?
  ```

  If you do not answer and press the Return key or type NO, SQL commits all changes made since the last COMMIT or ROLLBACK statement. If you answer YES to the prompt, SQL returns you to the SQL prompt.

OpenVMS VAX OpenVMS Alpha

- Typing Ctrl/Z is the same as issuing the EXIT statement for OpenVMS.♦

Digital UNIX

- Typing Ctrl/D is the same as issuing the EXIT statement for Digital UNIX.

  However if you define the RDB_IGNOREEOF configuration parameter in the .dbsrc configuration file, you can only exit interactive SQL by typing EXIT.♦

## EXPORT Statement

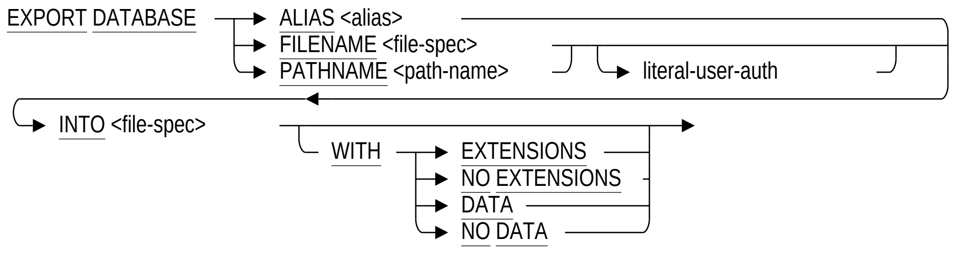
Makes a copy of a database in an intermediate form. Use the IMPORT statement to rebuild an Oracle Rdb database from the interchange file (.rbr file extension) created by the EXPORT statement.

You use the EXPORT statement with the IMPORT statement to make changes to Oracle Rdb databases that cannot be made any other way. The EXPORT statement unloads a database to an .rbr file. The IMPORT statement creates the database again with the changes that are both allowed and not allowed through ALTER statements. See the IMPORT Statement for more information.

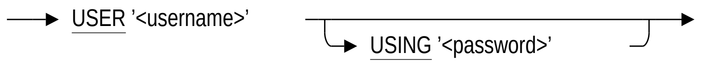
### Environment

You can use the EXPORT statement in interactive SQL only.

### Format



literal-user-auth =



### Arguments

**ALIAS alias**
**FILENAME file-spec**
**PATHNAME path-name**
Specifies the source database files to be written to an .rbr file.

- The ALIAS argument specifies the alias of an already attached database. If the database you want to export is already attached, specifying ALIAS avoids the overhead of a second attach to the database and the locking that attach entails.

- The FILENAME and PATHNAME arguments both identify the database root file associated with the database. If you specify a repository path name, the path name indirectly specifies the database root file. Because the EXPORT statement does not change any definitions in the repository, the effect of the PATHNAME and FILENAME arguments is the same.

OpenVMS VAX OpenVMS Alpha The PATHNAME argument can be specified only on OpenVMS platforms. ♦

**literal-user-auth**
Specifies the user name and password for access to databases, particularly remote database.

This literal lets you explicitly provide user name and password information in the EXPORT statement.

**USER 'username'**
Defines a character string literal that specifies the operating system user name that the database system uses for privilege checking.

**USING 'password'**
Defines a character string literal that specifies the user's password for the user name specified in the USER clause.

**INTO file-spec**
Specifies the name for the .rbr file the EXPORT statement creates. Optionally, the file specification can include a device and directory specification.

**WITH EXTENSIONS**
**WITH NO EXTENSIONS**
Specifies whether or not the .rbr file created by the EXPORT statement includes extensions that are compatible only with Oracle Rdb Version 3.0 or higher database systems. The default is WITH EXTENSIONS.

When you specify the WITH NO EXTENSIONS option, the resulting interchange (.rbr) file contains only the definitions of the domains, the tables, and indexes. Indexes are converted to sorted indexes and are minus storage maps. The following conversions take place for domains:

- TINYINT data types are converted to SMALLINT data types
- DATE ANSI, TIMESTAMP, and TIME data types are converted to DATE VMS data types

In addition, all null values are converted to the columns' missing value or default to a data type specific missing value. For example, null numeric values are replaced by zeros and null character values are replaced by blanks.

When you specify the WITH NO EXTENSIONS option, many features of Oracle Rdb databases are not exported. For example, storage areas, storage maps, triggers, collating sequences, functions, modules, and outlines are not backed up when you specify the WITH NO EXTENSIONS argument.

**Note**

OpenVMS VAX OpenVMS Alpha

The WITH NO EXTENSIONS option is not compatible with CDD/Repository databases (CDD$DATABASE.RDB). If you attempt to export a CDD$DATABASE.RDB database, SQL issues an error message stating that the WITH NO EXTENSIONS option is not valid for CDD/Repository databases. ♦

**WITH DATA**
**WITH NO DATA**
Specifies whether the .rbr file created by the EXPORT statement includes the data and metadata contained in the database, or the metadata only. The default is WITH DATA.

When you specify the WITH NO DATA option, the EXPORT statement copies metadata, but not the data, from a source database to an .rbr file. Use the IMPORT statement to generate an empty database whose metadata is identical to that of the source database.

**Note**

OpenVMS VAX OpenVMS Alpha

The WITH NO DATA option is not compatible with CDD/Repository databases (CDD$DATABASE.RDB). If you attempt to export a CDD$DATABASE.RDB database, SQL issues an error message stating that the WITH NO DATA option is not valid for CDD/Repository databases. ♦

## Usage Notes

- Before using EXPORT and IMPORT statements, be sure that the database was backed up in case either the EXPORT or IMPORT statement fails. Use the RMU Backup command.
- For information about how SQL handles character set information when you export and import a database, see the IMPORT Statement.
- You cannot use the EXPORT statement on a database that has over 65,535 segments in one segmented string field. However, the RMU Backup command does not have this limitation on the number of segments within a segmented string.
- The EXPORT statement does not check for a corrupt database. If the database is corrupt, EXPORT and IMPORT statements create a valid database, but the contents of that database may not be identical to the original.
- You need read access to all the tables in the database to back up the database with the EXPORT statement.
- The EXPORT statement does not allow Oracle Rdb system domains in user-defined tables. If the database names Oracle Rdb system domains in user-defined tables, Oracle Rdb returns an error message, and the export operation does not complete.
- If you use the ALTER DATABASE statement to set OPEN IS MANUAL on a database, you cannot export that database if it is closed.
- See the *Oracle Rdb7 Guide to Database Maintenance* for a complete discussion of when to use the IMPORT, EXPORT, and ALTER DATABASE statements.
- It is not possible to export a database using the WITH NO EXTENSIONS clause if it contains INTERVAL domains. Oracle Rdb recommends either removing the offending domains and related columns (see the DROP DOMAIN Statement) or performing the EXPORT operation without including the WITH NO EXTENSIONS clause.

- Normally, during an export operation, the Oracle Rdb interchange file (.rbr), which uses the Record Management Services (RMS) default extent, will extend for every 3 blocks the .rbr file grows in size. To prevent this, define the following SET statement to change the process default RMS extent quantity:

  ```
  $ SET RMS_DEFAULT/EXTEND_QUANTITY=30000
  ```

  Now, rather than "extending" the .rbr file for every 3 blocks (which involves many extend operations), the RMS extend is only invoked *once* per 30,000 blocks. By specifying a larger value for the file extend parameter, the run time of the export operation can be significantly improved.
- Oracle Rdb does not support remote export between different versions of Oracle Rdb. You can successfully export a database only if the version number of the system from which you issue the EXPORT statement equals the version number of the database you are exporting.

## FETCH Statement

Advances a cursor to the next row of its result table and retrieves the values from that row. When used with a list cursor, the FETCH statement places the cursor on a specified position within a list and retrieves a portion of that list. When embedded in precompiled host language programs, the FETCH statement assigns the values from the row to host parameters. In interactive SQL, the FETCH statement displays the value of the row on the terminal screen.

### Environment

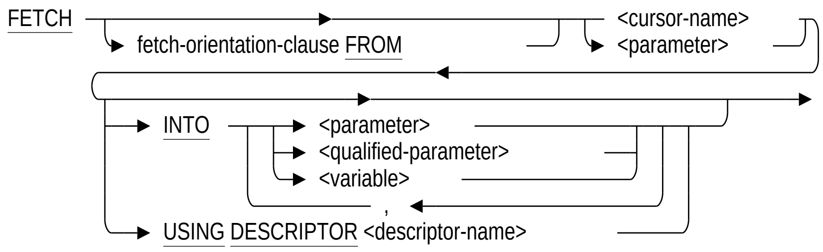
You can use the FETCH statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

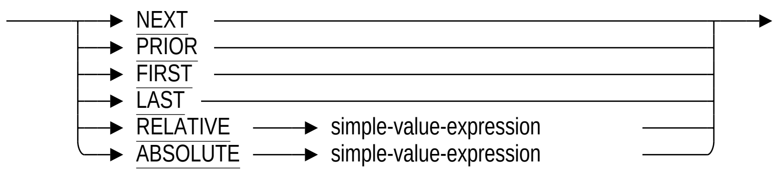
When you fetch a table cursor, you cannot use the INTO or USING clauses in interactive SQL. You must use either the INTO or USING clause in FETCH statements that are embedded in programs to be precompiled or SQL modules.

When you fetch a list cursor, you cannot use the USING clause in interactive SQL. With list cursors, you can only use the USING clause in FETCH statements that are embedded in programs to be precompiled or SQL modules.

### Format



fetch-orientation-clause =



## Arguments

**cursor-name**
**parameter**
Specifies the name of the cursor from which you want to retrieve a row. Use a parameter if the cursor referred to by the cursor name was declared at run time with a dynamic DECLARE CURSOR statement. Specify the parameter used for the cursor name in the dynamic DECLARE CURSOR statement.

You can use a parameter to refer to the cursor name only when the FETCH statement is accessing a dynamic cursor.

**fetch-orientation-clause FROM**
Specifies the specific segment of the list cursor to fetch. These options are available only if you specified the SCROLL option in the DECLARE CURSOR statement. The choices are:

- NEXT

  Fetches the next segment of the list cursor. This is the default.
- PRIOR

  Fetches the segment immediately before the current segment of the list cursor.
- FIRST

  Fetches the first segment of the list cursor.
- LAST

  Fetches the last segment of the list cursor.
- RELATIVE simple-value-expression

  Fetches the segment of the list cursor indicated by the value expression. For example, relative –4 would fetch the segment that is four segments prior to the current segment.

- ABSOLUTE simple-value-expression

  Fetches the segment of the list cursor indicated by the value expression. For example, absolute 4 would fetch the fourth segment of the list cursor.

**simple-value-expression**
Specifies either a positive or negative integer, or a numeric module language or host language parameter.

**INTO parameter**
**INTO qualified-parameter**
**INTO variable**
Specifies a list of parameters, qualified parameters (host structures), or variables to receive the values SQL retrieves from the row of the cursor's result table. The number of parameters or variables in the list must be the same as the number of values in the row. (If any of the parameters is a host structure, SQL counts the number of parameters in that structure when it compares the number of host parameters in the INTO clause with the number of values in the row.)

The data types of parameters and variables must be compatible with the values of the corresponding column of the cursor row.

**USING DESCRIPTOR descriptor-name**
Specifies the name of a descriptor that corresponds to an SQLDA. If you use the INCLUDE statement to insert the SQLDA into your program, the descriptor name is simply SQLDA.

An SQLDA is a collection of host language variables used only in dynamic SQL. In a FETCH statement, the SQLDA points to a number of parameters SQL uses to store values from the row. The number of parameters must match the number of columns in the row.

The data types of parameters must be compatible with the values of the corresponding column of the cursor row.

## Usage Notes

- You cannot use a FETCH statement for a cursor before you issue an OPEN statement for that cursor.

- An open cursor can be positioned:
  - Before the first row of its result table. When an OPEN statement executes, SQL positions the cursor before the first row. When a DELETE statement that refers to a cursor executes, SQL positions the cursor before the next row that follows the deleted row.
  - On a row of its result table (after a FETCH statement for any but the last row).
  - After the last row of its result table.

    When the table cursor is positioned on the last row, any FETCH or DELETE statement from the cursor positions the cursor after the last row.
- An error is generated and the SQLCODE status parameter or SQLCODE field of SQLCA is set to +100 and the SQLSTATE field is set to '02000' in the following situations:
  - If the current position of a cursor in a FETCH or FETCH NEXT statement is on or after the last row of its result table.
  - If a FETCH ABSOLUTE or FETCH RELATIVE statement tries to retrieve rows that are out of range.
  - If the current position of a cursor in a FETCH PRIOR statement is on or before the first row of its result table.
- If you attempt to fetch an element of a list into a target specification that is shorter than the element, the element will be truncated. The sixth element of the SQLERRD array of the SQLCA is set to the difference between the element and the target (the number of truncated bytes).
- Always use an indicator array when you use host language structures. For information about indicator arrays, see Section 2.2.19.2 or the *Oracle Rdb7 Guide to SQL Programming*.

  When SQL fetches a list cursor, the value of the indicator parameter shows if the segment is truncated. If no truncation occurs, the value of the indicator parameter is 0. If the list segment value is null, the value of the indicator parameter is –1. If the list segment is truncated, the SQLLEN stores the length of the untruncated segment.
- You can determine the length of the fetched segment by passing a VARCHAR or VARBYTE field in the SQLDA for the segment. SQL returns the length of the segment in the length field of these two data types.

- You must make sure you close the list cursor before fetching the next row of a table cursor. SQL does not issue an error message or warning if you forget to do so.

See Appendix B for more information on the SQLCA and Appendix C for more information on SQLSTATE.

## Examples

OpenVMS VAX OpenVMS Alpha

Example 1: Using a FETCH statement embedded in a PL/I program

This program fragment uses embedded DECLARE CURSOR, OPEN, and FETCH statements to retrieve and print the names and departments of managers. The FETCH statement fetches the rows of the result table and stores them in the parameters :FNAME, :LNAME, and :DNAME.

```
/* Declare the parameters: */
BEGIN DECLARE SECTION

DCL ID          CHAR(3);
DCL FNAME       CHAR(10);
DCL LNAME       CHAR(14);

END DECLARE SECTION

/* Declare the cursor: */
EXEC SQL DECLARE MANAGER CURSOR FOR
        SELECT E.FIRST_NAME, E.LAST_NAME, D.DEPARTMENT_NAME
                FROM EMPLOYEES E, DEPARTMENTS D
                WHERE E.EMPLOYEE_ID = D.MANAGER_ID ;

/* Open the cursor: */
EXEC SQL OPEN MANAGER;

/* Start a loop to process the rows of the cursor: */
DO WHILE (SQLCODE = 0);
        /* Retrieve the rows of the cursor
        and put the value in parameters: */
        EXEC SQL FETCH MANAGER INTO :FNAME, :LNAME, :DNAME;
        /* Print the values in the parameters: */
                        .
                        .
                        .
END;

/* Close the cursor: */
EXEC SQL CLOSE MANAGER;
```
♦

Example 2: Using a FETCH statement to display segments in a column of data type LIST

This interactive example uses a table cursor to retrieve a row that contains a list from the RESUMES table. The OPEN statement positions the cursor on the first segment of the list in the RESUME column, and subsequent FETCH statements retrieve successive segments of that list.

```
SQL> DECLARE TBLCURSOR2 CURSOR FOR SELECT EMPLOYEE_ID, RESUME
cont> FROM RESUMES;
SQL> DECLARE LSTCURSOR2 LIST CURSOR FOR SELECT RESUME
cont> WHERE CURRENT OF TBLCURSOR2;
SQL> OPEN TBLCURSOR2;
SQL> FETCH TBLCURSOR2;
  00164
SQL> OPEN LSTCURSOR2;
SQL> FETCH LSTCURSOR2;
 RESUME
 This is the resume for 00164
SQL> FETCH LSTCURSOR2;
 RESUME
 Boston, MA
SQL> FETCH LSTCURSOR2;
 RESUME
 Oracle Corporation
SQL> FETCH LSTCURSOR2;
 RESUME
%RDB-E-STREAM_EOF, attempt to fetch past end of record stream
SQL> CLOSE LSTCURSOR2;
SQL> SELECT * FROM RESUMES;
  EMPLOYEE_ID    RESUME
 00164                  72:2:3
1 row selected
SQL> CLOSE TBLCURSOR2;
SQL> COMMIT;
```

Example 3: Using a scrollable list cursor to fetch list data

This C program demonstrates the use of scrollable list cursors to read list data from the sample personnel database using the FETCH statement. The list data being read is from the RESUME column of the RESUMES table in personnel. Note that the RESUME is divided into three segments in this order:

1. A line including the employee's name: "This is the resume for Alvin Toliver"
2. A line stating where the employee lives: "Boston, MA"

3. A line stating where the employee works: “Oracle Corporation”

```
#include stdio
#include descrip

/* Declare parameters for error handling by including the SQLCA. */

EXEC SQL INCLUDE SQLCA;

/* Error-handling section. */

dump_error( )
{

short           errbuflen;
char            errbuf[ 1024 ];
struct          dsc$descriptor_s    errbufdsc;

errbufdsc.dsc$b_class = DSC$K_CLASS_S;
errbufdsc.dsc$b_dtype = DSC$K_DTYPE_T;
errbufdsc.dsc$w_length = 1024;
errbufdsc.dsc$a_pointer = &errbuf;

    if (SQLCA.SQLCODE != 0)
    {
        printf( "SQLCODE = %d\n", SQLCA.SQLCODE );
       SQL$GET_ERROR_TEXT( &errbufdsc, &errbuflen );
     errbuf[ errbuflen ] = 0;
     printf("%s\n", &errbuf );
    }
}
main()
{

/* Attach to the personnel database. */

EXEC SQL DECLARE ALIAS FILENAME personnel;

/* Declare variables. */

short two_s;

long  two_l;

char    blob[8];
char    emp_id[6];
char    seg2[ 81 ];

/* Declare a table cursor. */

    exec sql declare resumes_cursor table cursor for
    select employee_id, resume from resumes where employee_id = ’00164’;

/* Declare a read-only scrollable list cursor to fetch the RESUME column. */

    exec sql declare resume_list_cursor read only scrollable list cursor for
    select resume where current of resumes_cursor;

/* Open the table cursor. */
```

```
    exec sql open resumes_cursor;
    dump_error();

/* Place the first value in the table cursor (00164) into the emp_id parameter,
and the resume data into the blob parameter. */

    exec sql fetch resumes_cursor into :emp_id, :blob;
    dump_error();

/* Open the scrollable list cursor. */

exec sql open resume_list_cursor;
dump_error();

/*  Begin to use the FETCH statement to read desired lines from the resume.
    If an attempt is made to retrieve a segment that is out of range, the
    program prints an error message.
*/

exec sql fetch last from resume_list_cursor into :seg2;
printf("FETCH LAST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch next from resume_list_cursor into :seg2;
printf("FETCH NEXT segment returned: %s\n", seg2 );
dump_error();

exec sql fetch first from resume_list_cursor into :seg2;
printf("FETCH FIRST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch next from resume_list_cursor into :seg2;
printf("FETCH NEXT segment returned: %s\n", seg2 );
dump_error();

exec sql fetch next from resume_list_cursor into :seg2;
printf("FETCH NEXT segment returned: %s\n", seg2 );
dump_error();

exec sql fetch relative -2 from resume_list_cursor into :seg2;
printf("FETCH RELATIVE -2 segment returned: %s\n", seg2 );
dump_error();

exec sql fetch first from resume_list_cursor into :seg2;
printf("FETCH FIRST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch relative 2 from resume_list_cursor into :seg2;
printf("FETCH RELATIVE 2 segment returned: %s\n", seg2 );
dump_error();

exec sql fetch last from resume_list_cursor into :seg2;
printf("FETCH LAST segment returned: %s\n", seg2 );
dump_error();

exec sql fetch prior from resume_list_cursor into :seg2;
printf("FETCH PRIOR segment returned: %s\n", seg2 );
dump_error();
```

```
exec sql fetch ABSOLUTE 1 from resume_list_cursor into :seg2;
printf("FETCH ABSOLUTE 1 segment returned: %s\n", seg2 );
dump_error();

exec sql fetch relative 2 from resume_list_cursor into :seg2;
printf("FETCH RELATIVE 2 segment returned: %s\n", seg2 );
dump_error();

two_s = 2;
exec sql fetch ABSOLUTE :two_s from resume_list_cursor into :seg2;
printf("FETCH ABSOLUTE :two_s segment returned: %s\n", seg2 );
dump_error();

two_l = 2;
exec sql fetch ABSOLUTE :two_l from resume_list_cursor into :seg2;
printf("FETCH ABSOLUTE :two_1 segment returned: %s\n", seg2 );
dump_error();

    exec sql fetch RELATIVE :two_l from resume_list_cursor into :seg2;
    printf("FETCH RELATIVE :two_1 segment returned: %s\n", seg2 );
    dump_error();

    exec sql rollback;
}
```

This preceding program is not available as a sample program with this release of Oracle Rdb. However, the following example shows the output from the program:

```
FETCH LAST segment returned: Oracle Corporation
FETCH NEXT segment returned: Oracle Corporation
SQLCODE = 100
%SQL-W-NOTFOUND, No rows were found for this statement
FETCH FIRST segment returned: This is the resume for Alvin Toliver
FETCH NEXT segment returned: Boston, MA
FETCH NEXT segment returned: Oracle Corporation
FETCH RELATIVE -2 segment returned: This is the resume for Alvin Toliver
FETCH FIRST segment returned: This is the resume for Alvin Toliver
FETCH RELATIVE 2 segment returned: Oracle Corporation
FETCH LAST segment returned: Oracle Corporation
FETCH PRIOR segment returned: Boston, MA
FETCH ABSOLUTE 1 segment returned: This is the resume for Alvin Toliver
FETCH RELATIVE 2 segment returned: Oracle Corporation
FETCH ABSOLUTE :two_s segment returned: Boston, MA
FETCH ABSOLUTE :two_1 segment returned: Boston, MA
FETCH RELATIVE :two_1 segment returned: Boston, MA
SQLCODE = -1
%RDB-F-SEGSTR_EOF, attempt to fetch past the end of a segmented string
-RDMS-E-FETRELATIVE, fetch relative (2) causes reference out of range 1..3
```

## FOR Control Statement

Executes an SQL statement for each row of a query expression.

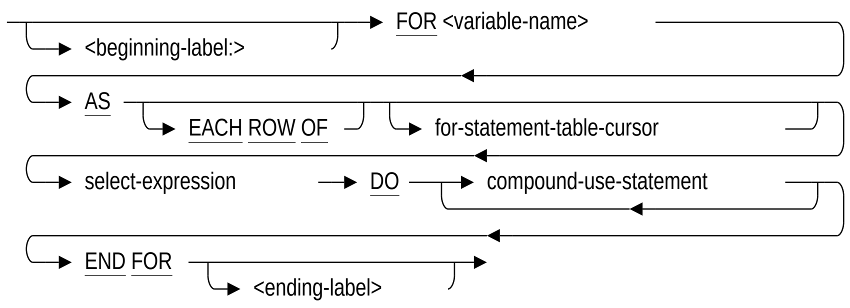
### Environment

You can use the FOR control statement in a compound statement of a multistatement procedure:

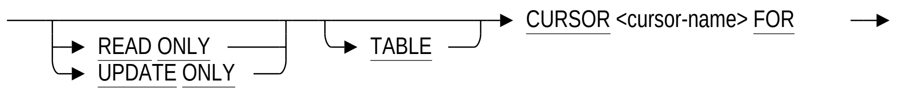
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

for-statement =



for-statement-table-cursor =



### Arguments

**beginning-label:**
Assigns a name to the FOR statement.

A beginning label used with the LEAVE statement lets you perform a controlled exit from a FOR loop. A named FOR loop is called a **labeled FOR loop statement**. If you include an ending label, it must be identical to its corresponding beginning label. A beginning label must be unique within the procedure in which the label is contained.

**FOR variable-name**
Specifies a name for a record consisting of a field for each named column of the FOR loop select expression. Each field in the record contains the data represented by each column name in each row of the select expression result table.

The variable name lets you reference a field in the compound-use-statement argument, for example: variable-name.column-name.

**AS EACH ROW OF for-statement-table-cursor**
Creates a result table with a specified cursor.

The optional naming of a cursor lets you use positioned data manipulation language statements in the DO clause of a FOR loop.

**AS EACH ROW OF select-expression**
Creates a simple result table.

After SQL creates the result table from the select expression, the DO clause executes a set of SQL statements (compound-use-statement) for each result table row.

**DO compound-use-statement**
Executes a block of SQL statements for each row of the select expression result table.

**END FOR ending-label**
Marks the end of a FOR loop. If you choose to include the optional ending label, it must match exactly its corresponding beginning label. An ending label must be unique within the procedure in which the label is contained.

The optional end-label argument makes the FOR loops of multistatement procedures easier to read, especially in very complex procedure blocks.

## Usage Notes

- The cursor name must be unique within the containing module.
- Reference to the cursor name is only valid inside this FOR statement.
- Variables are created at the beginning of the FOR statement and are destroyed at the end of the FOR statement.
- A FOR cursor loop executes the DO . . . END FOR body of the loop for each row fetched from the row set. Applications cannot use RETURNED_SQLCODE or RETURNED_SQLSTATE to determine if the FOR loop reached the end of the row set without processing any rows. Applications should use the GET DIAGNOSTICS ROW_COUNT statement after the END FOR clause to test for zero or more rows processed.

## Examples

Example 1: Using the FOR statement within an SQL module procedure

```
-- This procedure counts the employees of a given state who have had
-- a decrease in their salary during their employment.

PROCEDURE COUNT_DECREASED (SQLSTATE :state CHAR(2),
                                    :n_decreased INTEGER);

BEGIN
   DECLARE :last_salary INTEGER (2);
   SET n_decreased = 0;

   emp_loop:
    FOR :empfor
       AS EACH ROW OF
          SELECT * FROM EMPLOYEES emp WHERE STATE = :state
          DO
          SET :last_salary = 0;

           history_loop:
           FOR :salfor
              AS EACH ROW OF
                SELECT salary_amount FROM  salary_history s
                        WHERE s.EMPLOYEE_ID = :empfor.EMPLOYEE_ID
           DO
             IF SALARY_AMOUNT > :LAST_SALARY
               THEN
```

```
                    SET :N_DECREASED = :N_DECREASED +1 ;
                    LEAVE HISTORY_LOOP;
                 END IF;

                 SET :LAST_SALARY = :SALARY_AMOUNT;
               END FOR;
       END FOR;
    END;
```

## GET DIAGNOSTICS Statement

Extracts diagnostic information about the execution of the previous SQL statement.

The GET DIAGNOSTICS statement captures diagnostic information from an Oracle Rdb maintained data structure called the **diagnostics area**. In the ANSI/ISO SQL standard, the diagnostics area consists of two components: a single header area and an array of detail areas. Oracle Rdb extracts information only from the header component and the first element of the detail area (Exception 1):

- Header area

  Contains status information about rows and transactions, for example, the number of rows affected by an INSERT, UPDATE, or DELETE statement or the type of transaction that is active.

  See the statement-item-name argument for a complete list of the status information you can retrieve from the header area.

- Detail area (Exception 1)

  Contains diagnostic information that corresponds to the status that would be reported in the SQLSTATE or SQLCODE status parameter. The EXCEPTION . . . RETURNED_SQLSTATE argument retrieves the SQLSTATE status information from the detail area. The EXCEPTION . . . RETURNED_SQLCODE argument retrieves the SQLCODE status information from the detail area.

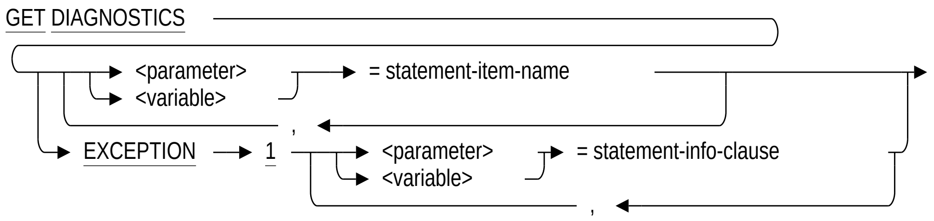
### Environment

You can use the GET DIAGNOSTICS statement only within the compound statement of a multistatement procedure:

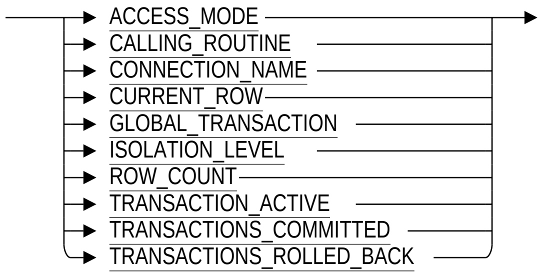
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a multistatement procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

## GET DIAGNOSTICS Statement

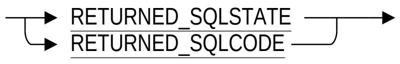
## Format



statement-item-name =



statement-info-clause =



## Arguments

**parameter = statement-item-name**
**variable = statement-item-name**
Retrieves information about the statement execution recorded in the diagnostics area and stores it in a simple target specification (a parameter or variable).

**statement-item-name**
Specifies the kind of diagnostic information you can retrieve about a previously executed SQL statement. You can gather the following diagnostic data:

- ACCESS_MODE returns the character string READ ONLY, READ WRITE, or BATCH UPDATE to indicate the type of transaction that is active. These character strings are of the CHAR data type. The argument also returns NONE when no transaction is active. See the SET TRANSACTION Statement for a description of transaction access modes.

- CALLING_ROUTINE returns a string of data type CHAR(31) of the name of the calling routine. If there is no name for the calling routine, spaces are returned.
- CONNECTION_NAME returns the current connection name.
- CURRENT_ROW returns the integer value for the number of rows that have been fetched by the innermost FOR control statement.
- GLOBAL_TRANSACTION returns an integer of 1 when a global transaction is active and an integer of 0 otherwise.
- ISOLATION_LEVEL returns the character string READ COMMITTED, REPEATABLE READ, or SERIALIZABLE to indicate the isolation level of a transaction. These character strings are of the CHAR data type. The argument also returns NONE when no transaction is active. See the SET TRANSACTION Statement for a description of transaction isolation levels.
- ROW_COUNT returns an integer for the number of rows affected by an INSERT, UPDATE, or DELETE statement.
- TRANSACTION_ACTIVE returns an integer of 1 when a transaction is active and an integer of 0 otherwise.
- TRANSACTIONS_COMMITTED returns an integer value for the number of transactions that have been committed during the processing of a multistatement procedure.
- TRANSACTIONS_ROLLED_BACK returns an integer value for the number of transactions that have been rolled back during the processing of a multistatement procedure.

**EXCEPTION 1 parameter = RETURNED_SQLSTATE**
**EXCEPTION 1 variable = RETURNED_SQLSTATE**
Retrieves diagnostics information equivalent to the value of SQLSTATE returned after execution of the previous SQL statement (other than a GET DIAGNOSTICS statement). The data type of the returned information is CHAR(5).

**EXCEPTION 1 parameter = RETURNED_SQLCODE**
**EXCEPTION 1 variable = RETURNED_SQLCODE**
Retrieves diagnostics information equivalent to the value of SQLCODE returned after execution of the previous SQL statement (other than a GET DIAGNOSTICS statement). The data type of the returned information is integer.

## Usage Notes

- The diagnostics area is cleared at the beginning of each multistatement procedure.
- You can use the GET DIAGNOSTICS statement only within the compound statement of a multistatement procedure.
- Because an exception causes a multistatement procedure to terminate immediately, RETURNED_SQLCODE or RETURNED_SQLSTATE only returns a warning message. If the procedure is successful, RETURNED_SQLCODE or RETURNED_SQLSTATE returns a success message.

## Examples

Example 1: Using a GET DIAGNOSTICS statement to retrieve row count

```
PROCEDURE increate_nh (SQLSTATE, :rows_affected INTEGER);
    BEGIN ATOMIC
        UPDATE salary_history
        SET    salary_amount = salary_amount * 1.05
        WHERE  salary_end IS NULL
          AND  employee_id IN (SELECT    employee_id
                               FROM      employees
                               WHERE     state = 'NH' );
        GET DIAGNOSTICS :rows_affected = ROW_COUNT;
    END;
```

Example 2: Using RETURNED_SQLSTATE

```
SQL> DECLARE :Y CHAR(5);
SQL> BEGIN
cont> SET :Y = 'Hello';
cont> GET DIAGNOSTICS EXCEPTION 1 :Y = RETURNED_SQLSTATE;
cont> END;
SQL> PRINT :Y;
 Y
 00000
SQL>
```

Example 3: Using RETURNED_SQLCODE

```
SQL> DECLARE :X INTEGER;
SQL> BEGIN
cont> SET :X = 100;
cont> GET DIAGNOSTICS EXCEPTION 1 :X = RETURNED_SQLCODE;
cont> END;
SQL> PRINT :X;
           X
           0
```

Example 4: Returning the current connection name

```
SQL> CONNECT TO 'ATTACH FILENAME mf_personnel' AS 'my_connection';
SQL> DECLARE :conn_name VARCHAR(20);
SQL> BEGIN
cont>    GET DIAGNOSTICS :conn_name = CONNECTION_NAME;
cont> END;
SQL> PRINT :conn_name;
 CONN_NAME
 my_connection
```

## GRANT Statement

Creates or adds privileges to an entry to the Oracle Rdb access privilege set, called the **access control list (ACL)**, for a database, table, view, column, module, or external routine. Each entry in an ACL consists of an identifier and a list of privileges assigned to the identifier:

- Each identifier specifies a user or a set of users.
- The list of privileges specifies which operations that user or user group can perform on the database, table, view, column, module, or external routine.

When a user tries to perform an operation on a database, SQL reads the associated ACL from top to bottom, comparing the identifier of the user with each entry. As soon as SQL finds the first match, it grants the rights listed in that entry and stops the search. All identifiers that do not match a previous entry "fall through" to the entry [*,*] (equivalent to the SQL keyword PUBLIC). If no entry has the identifier [*,*], then users with unmatched identifiers are denied all access to the database, table, view, column, module, or external routine.

For this reason, both the entries and their order in the list are important.

Under the Oracle Rdb default protection scheme, when you create a new database, table, view, module, or external routine, you get all access rights to that object, including DBCTRL. All other users of that object are given no access rights to it. For any tables or views created under the Oracle Rdb default protection scheme, the creator of the table or view receives all the access rights to the object, including DBCTRL, and all other users receive no access rights to the object.

The DBCTRL access right enables an object's creator to grant DBCTRL to other users. See the Usage Notes for information on how you can tailor the default protection for any new tables that you create within a database.

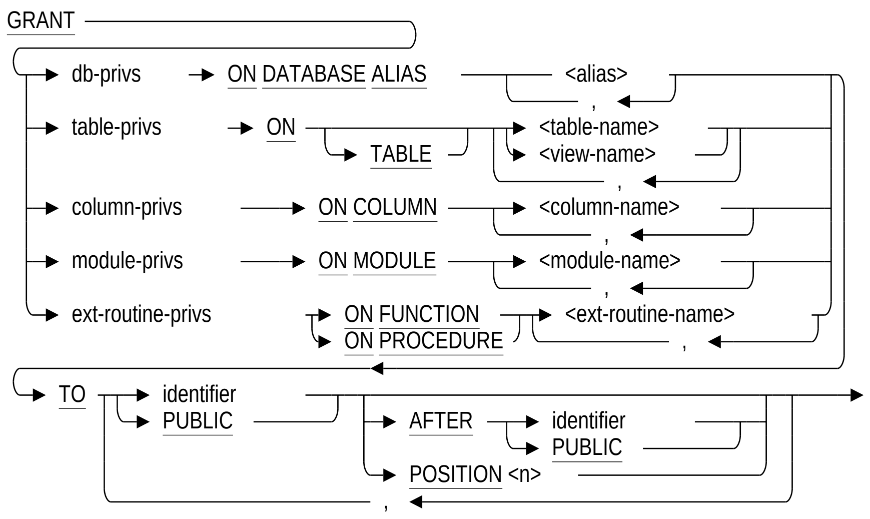
To remove privileges from or entirely delete an entry to the Oracle Rdb access privilege set for a database, table, column, module, or external routine, see the REVOKE Statement.

## Environment

You can use the GRANT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

## Format



## GRANT Statement

db-privs =

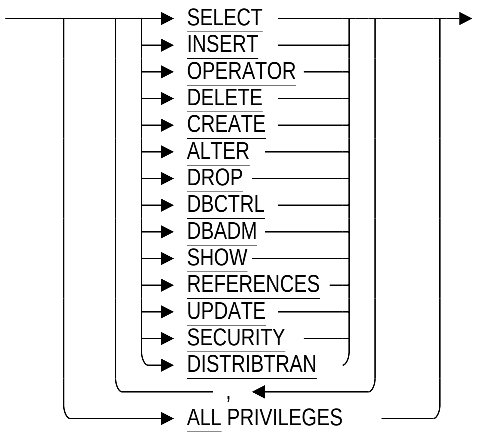
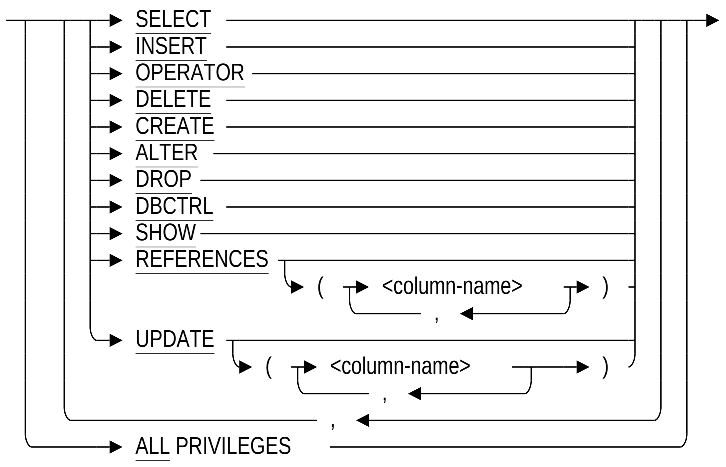
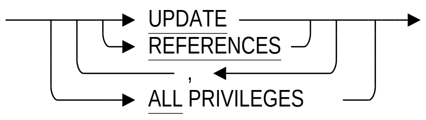


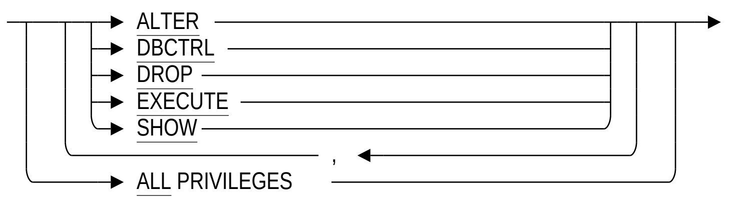
table-privs=



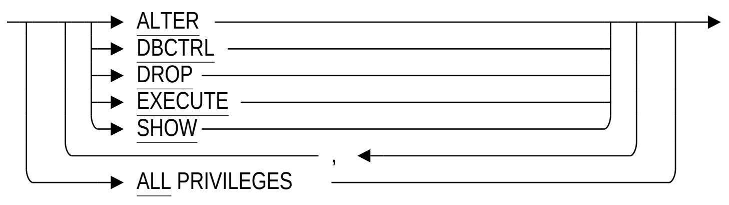
column-privs=



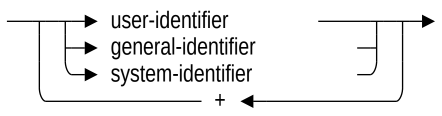
module-privs =



ext-routine-privs =



identifier =



## Arguments

**db-privs**
**table-privs**
**column-privs**
**module-privs**
**ext-routine-privs**

Specifies the list of privileges you want to add to an existing ACL entry or create in a new one. The operations permitted by a given privilege keyword differ, depending on whether you granted it for a database, table, column, module, or external routine.

Table 7–1 lists the privilege keywords and their meanings for databases, tables, columns, modules, and external routine.

**Table 7–1 SQL Privileges for Databases, Tables, Columns, Modules, and External Routines**

| Privilege | For the Access Privilege Set of a Database, Grants the Privilege to: | For the Access Privilege Set of a Table, Column, View, Module, or External Routine, Grants the Privilege to: |
|---|---|---|
| ALTER | Change database parameters or change a domain. | Alter the table, index, or storage map. Alter a module or external routine. Does not apply to column privileges. |
| CREATE | Create a catalog, schema, table, domain, collating sequence, storage area, external routine, or module. | Create a view, trigger, index, storage map, or outline that uses a table. Does not apply to column privileges. |
| DBADM | Perform any data manipulation or data definition operation on any named object. Override many database privileges. | Not applicable, but syntactically allowed. |
| DBCTRL | Create, delete, or modify an access privilege set entry for the database. | Create, delete, or modify an access privilege set entry for the table, module, or external routine. Does not apply to column privileges. |
| DELETE | Delete data from a table defined in the database. | Delete data from a table. Does not apply to column privileges. |
| DISTRIBTRAN | Run a distributed (two-phase commit protocol) transaction against the database. | Not applicable. |
| DROP | Delete a catalog, schema, domain, collating sequence, or path name. | Delete the table, delete a view, index or outline that uses a table. Delete a column, constraint, trigger or storage map. Delete a module or external routine. |

(continued on next page)

**Table 7–1 (Cont.) SQL Privileges for Databases, Tables, Columns, Modules, and External Routines**

| Privilege | For the Access Privilege Set of a Database, Grants the Privilege to: | For the Access Privilege Set of a Table, Column, View, Module, or External Routine, Grants the Privilege to: |
|---|---|---|
| EXECUTE | Not applicable. | Allow the execution of a module or external routine. Does not apply to column or table privileges. |
| INSERT | Store data in a table defined in the database. | Store data in the table. Does not apply to column privileges. |
| REFERENCES | Not applicable, but syntactically allowed. | Define constraints that refer to data in a table or column. |
| SECURITY | Override many database privileges. | Not applicable. |
| SELECT | Attach to a database and read data from a table defined in the database. | Read data from a table. Does not apply to column privileges. |
| SHOW | Not applicable. Syntactically allowed, but not implemented. Reserved for future versions. | Not applicable. Syntactically allowed, but not implemented. Reserved for future versions. |
| UPDATE | Update data in a table defined in the database. | Update data in a table or column. |

Privileges on a column are determined by the privileges defined for the table combined with those specified for the specific column ACL.

The SELECT privilege is a prerequisite for all other data manipulation privileges, except UPDATE and REFERENCES. If you do not grant the SELECT privilege, you effectively deny SELECT, INSERT, and DELETE privileges, even if they are specified in the privilege list. It is not possible for you to deny yourself the SELECT privilege.

For the SELECT, INSERT, UPDATE, and DELETE data manipulation privileges, SQL checks the ACL for the database and for the individual table before allowing access to a specific table. For example, if you are granted SELECT privilege for the EMPLOYEES table, you are not able to select rows from the table unless you also have SELECT privilege for the database that contains the EMPLOYEES table.

A user with the UPDATE privilege on the table automatically receives the UPDATE privilege on all columns in the table. To update a column, you must have the UPDATE privilege either for the column or the table. However, you can restrict the UPDATE privileges by defining them only on specific columns you want users to be able to update, and by removing the UPDATE privilege from the table entry.

You can modify the data in a column only with the UPDATE privilege on the column and the SELECT privilege on the database.

The REFERENCES privilege lets you define a constraint for a database with ANSI/ISO-style privileges. For a database with ACL-style privileges, you need the CREATE privilege to define a constraint.

You cannot deny yourself the DBCTRL privilege for a database or table that you create. This restriction may cause GRANT statements to fail when you might expect them to work.

For instance, suppose an ACL has no entry for PUBLIC. The following GRANT statement fails because it creates an entry for PUBLIC at the top of the ACL that does not include the DBCTRL privilege, effectively denying DBCTRL to all other entries on the list, including the owner:

```
SQL> GRANT SELECT, INSERT ON EMPLOYEES TO PUBLIC;
%RDB-E-NO_PRIV, privilege denied by database facility
```

**ALL PRIVILEGES**
Specifies that SQL should grant all privileges in the ACL entry.

**ON DATABASE ALIAS alias**
**ON TABLE table-name**
**ON TABLE view-name**
**ON COLUMN column-name**
**ON MODULE module-name**
**ON FUNCTION ext-routine-name**
**ON PROCEDURE ext-routine-name**
Specifies whether the GRANT statement applies to ACLs for databases, tables, columns, views, external routines, or modules. You can specify a list of names for any form of the ON clause. You must qualify a column name with at least the associated table name. In the following example, the table name SALARY_ HISTORY qualifies the EMPLOYEE_ID column name:

```
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.EMPLOYEE_ID TO PUBLIC;
```

**TO identifier**
**TO PUBLIC**
Specifies the identifiers for the new or modified ACL entry. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

You can specify three types of identifiers:

- User identifiers
- General identifiers — (OpenVMS only)
- System-defined identifiers — (OpenVMS only)

OpenVMS VAX OpenVMS Alpha

You can specify more than one identifier by combining them with plus signs (+). Such identifiers are called multiple identifiers. They identify only those users who are common to all the groups defined by the individual identifiers. Users who do not match all the identifiers are not controlled by that entry.

For instance, the multiple identifier SECRETARIES + INTERACTIVE specifies only members of the group defined by the general identifier SECRETARIES that are interactive processes. It does not identify members of the SECRETARIES group that are not interactive processes. ♦

The following arguments briefly describe the three types of identifiers. For more information about identifiers, see your operating system documentation.

**user-identifier**
Identifies each user on the system.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the user identifier consists of the standard OpenVMS user identification code (UIC), a group name and a member name (user name). The group name is optional. The user identifier can be in either numeric or alphanumeric format. The following are all valid user identifiers that could identify the same user:

```
K_JONES
[SYSTEM3, K_JONES]
[341,311] ♦
```

Digital UNIX

On Digital UNIX, a user identifier consists of the name of a group and the standard Digital UNIX user identifier (UID). The group name is optional. The user identifier must be in alphanumeric form. On Digital UNIX, users can belong to more than one group. The following are valid user identifiers that could identify the same user:

```
k_jones
[system3, k_jones] ♦
```

You can use the asterisk (*) wildcard character as part of a user identifier. For example, if you want to specify all users in a group, you can enter [system3, *] as the identifier.

When Oracle Rdb creates a database, it automatically creates an ACL entry with the identifier [*,*], which specifies the privileges given to all users on the system.

You cannot use more than one user identifier in a multiple identifier.

OpenVMS VAX OpenVMS Alpha

**general-identifier**
Identifies groups of users on the system and are defined by the OpenVMS system manager in the system rights database. The following are possible general identifiers:

DATAENTRY
SECRETARIES
MANAGERS ♦

OpenVMS VAX OpenVMS Alpha

**system-identifier**
System-defined identifiers are automatically defined by the system when the rights database is created at system installation time. System-defined identifiers are assigned depending on the type of login you execute. The following are all valid system-defined identifiers:

BATCH
NETWORK
INTERACTIVE
LOCAL
DIALUP
REMOTE ♦

**AFTER identifier**
**AFTER PUBLIC**
**POSITION n**
Specifies the position of the entry within the ACL to be modified or created.

With the AFTER or POSITION argument, you can specify the position in the list after which SQL searches for an ACL entry with an identifier that matches the one specified in the TO clause of the GRANT statement.

Digital UNIX

On Digital UNIX, because users can belong to more that one group, SQL considers entries as matching only when the group and user name in both entries are identical. ♦

Following are specifics about the AFTER and POSITION arguments:

- In the AFTER argument, the identifier specifies the entry in the ACL after which SQL begins its search for the entry to be modified or created. If none of the entries in the ACL has an identifier that matches the identifier specified in the AFTER argument, SQL generates an error and the statement fails.

  Starting *after* the entry specified by the identifier in the AFTER argument, SQL searches the entries in the ACL. If an entry has an identifier that matches the identifier specified by the TO clause of the GRANT statement, SQL creates a new entry that contains only those privileges specified in the GRANT statement. SQL retains only the entry appearing first in the ACL, and deletes any entries with duplicate identifiers.

  If none of the entries has an identifier that matches the identifier specified by the TO clause of the GRANT statement, SQL creates a new ACL entry immediately following the identifier specified in the AFTER argument.

  Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

- In the POSITION argument, the integer specifies the earliest relative position in the ACL of the entry to be modified or created.

  Starting *with* the position specified by the POSITION argument, SQL searches the entries in the ACL. If an entry has an identifier that matches the identifier specified by the TO clause of the GRANT statement, SQL creates a new entry that contains only those privileges specified in the GRANT statement. SQL retains only the entry appearing first in the ACL, and deletes any entries with duplicate identifiers.

  If none of the entries has an identifier that matches the identifier specified by the TO clause of the GRANT statement, SQL creates a new entry for that identifier at the relative position specified in the POSITION argument (even if an entry before the position at which SQL began its search had an identifier that matched).

  If you specify a position higher than the number of entries in the list, SQL places the entry last in the ACL. For example, if you specify position 12 and there are only 10 entries in the list, the new entry is placed in position 11 and given that position number.

- If you omit the AFTER or POSITION argument, SQL searches the entire ACL for an identifier list that matches the one specified in the TO clause of the GRANT statement. If it finds a match, it modifies the ACL entry by adding those privileges specified in the privilege list that are not already present. If there is no match, SQL creates a new entry at the beginning of the ACL.

Digital UNIX

On Digital UNIX, if the ACL contains an entry for a user, and the GRANT statement specifies the same user, but a different group, SQL adds the new entry. For example, and ACL could contain the following entries:

[eng_grp, janasik]
[users, janasik]
[*, janasik] ♦

## Usage Notes

- Additions and changes to ACLs do not take effect until you attach to the database again, even though those changes are displayed by the SHOW PROTECTION and SHOW PRIVILEGES statements. Additions and changes to ACLs do not take effect for other users until they attach to the database again.
- You must attach to all databases that you refer to in a GRANT statement. If you use the default alias, you must use the alias RDB$DBHANDLE to work with database ACLs.
- You can use the GRANT statement to modify existing ACL entries or create new ones.

  To modify an existing ACL entry, specify the same identifier in the TO clause as is in the existing entry.

  To create a new ACL entry, specify an identifier that is not already part of an entry.

Digital UNIX

- On Digital UNIX, if you do not specify a group, SQL displays the group specified in the operating system's password file, as shown in the following example:

```
SQL>  GRANT ALL ON DATABASE ALIAS RDB$DBHANDLE TO janasik;
SQL> SHOW PROTECTION ON DATABASE RDB$DBHANDLE
Protection on Alias RDB$DBHANDLE
```

```
(IDENTIFIER=[users,janasik],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
      CREATE+ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[users,heleng],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
      CREATE+ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[*,*],ACCESS=NONE)                  ♦
```

- When you create new tables and views, they have a PUBLIC access of NONE by default.
- OpenVMS VAX OpenVMS Alpha To override PUBLIC access for newly created tables, define an identifier with the name DEFAULT in the system privileges database. The access privileges given to this identifier on your database are then assigned to PUBLIC for any newly created tables and views.

  You might want to assign the SELECT and UPDATE privileges to the database with alias TEST1. For example:

```
SQL> ATTACH 'ALIAS TEST1 FILENAME personnel';
SQL> SHOW PROTECTION ON DATABASE TEST1.
Protection on Alias TEST1
    (IDENTIFIER=[dbs,smallwood],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
     CREATE+ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY)
    (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> GRANT SELECT, UPDATE ON DATABASE ALIAS TEST1
cont> TO DEFAULT;
```

  You must commit and disconnect the transaction to make the change in protection occur.

```
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
```

  The protection on existing tables in the database is not changed, but any new tables that you define receive the protection specified by the DEFAULT identifier. In this example, the owner (SMALLWOOD) receives all the access privileges to the new table TABLE1, and all other users receive the SELECT and UPDATE access privileges specified by the DEFAULT identifier.

```
SQL> ATTACH 'ALIAS TEST1 FILENAME personnel';
SQL> SET TRANSACTION READ WRITE;
SQL> CREATE TABLE TEST1.TABLE1
cont>       (LAST_NAME_DOM CHAR(5),
cont>        YEAR_DOM SMALLINT);
SQL> SHOW PROTECTION ON TEST1.TABLE1;
Protection on Table TABLE1
    (IDENTIFIER=[dbs,smallwood],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
      CREATE+ALTER+DROP+DBCTRL+DBADM+REFERENCES+SECURITY)
    (IDENTIFIER=[*,*],ACCESS=SELECT+UPDATE)
```

The DEFAULT identifier is typically present on an OpenVMS system because the DEFAULT account is always present and cannot be removed. However, it is possible to remove the DEFAULT identifier associated with that account. If the DEFAULT identifier was removed from your system, Oracle Rdb returns an error message.

```
SQL> GRANT INSERT ON DATABASE ALIAS TEST1 to DEFAULT;
%SYSTEM-F-NOSUCHID, unknown rights identifier    ♦
```

- The DBADM and SECURITY privileges are the two Oracle Rdb role-oriented privileges. Users with these privileges can override ACLs for some objects to perform certain system-level operations. Oracle Rdb role-oriented privileges are limited to the database in which they are granted.

  The two role-oriented database privileges cannot override each other. (For example, DBADM privilege does not override SECURITY privilege.)

  A user having one of these role-oriented privileges may be implicitly granted certain other Oracle Rdb privileges. An implicit privilege is a privilege granted as a result of an override; the user operates as if the user actually holds the privilege, but the privilege is not explicitly granted and stored in the ACL for the object.

  Table 7–2 shows which Oracle Rdb privileges can be overridden by the Oracle Rdb DBADM and SECURITY database privileges and the OpenVMS SYSPRV, BYPASS, and READALL privileges. For each table entry, the question is whether users with the Oracle Rdb or OpenVMS privilege specified in the columns at the top of the table implicitly receive the access rights associated with the Oracle Rdb privilege in the first column of the table. (The Digital UNIX operating system does not support the concept of system privileges.)

  For example, the *Y* for the first entry in the table shows that the Oracle Rdb DBADM privilege overrides the Oracle Rdb ALTER privilege, and the *N* in the second entry shows that the Oracle Rdb SECURITY privilege does not override the Oracle Rdb ALTER privilege. An *N/A* entry in the table indicates that a privilege cannot override itself.

**Table 7–2 Privilege Override Capability**

| | Database Privilege | | OpenVMS Privilege | | |
|---|---|---|---|---|---|
| **Privilege** | **DBADM** | **SECURITY** | **SYSPRV** | **BYPASS** | **READALL** |
| ALTER | Y | N | Y | Y | N |
| CREATE | Y | N | Y | Y | N |
| DBADM | N/A | N | Y | N | N |
| DBCTRL | Y | Y | Y | N | N |
| DELETE (database) | Y | Y | Y | Y | N |
| DELETE (table) | Y | N | Y | Y | N |
| DISTRIBTRAN | Y | N | Y | Y | N |
| DROP | Y | N | Y | Y | N |
| EXECUTE | Y | Y | N | N | N |
| INSERT (database) | Y | Y | Y | Y | N |
| INSERT (table) | Y | N | Y | Y | N |
| REFERENCES | Y | N | Y | Y | N |
| SECURITY | N | N/A | N | N | N |
| SELECT (database) | Y | Y | Y | Y | Y |
| SELECT (table) | Y | N | Y | Y | Y |
| SHOW | Y | N | Y | Y | Y |
| UPDATE (database) | Y | Y | Y | Y | N |
| UPDATE (table) | Y | N | Y | Y | N |

- Users with the DBADM database privilege can perform any data definition or data manipulation operation on any named object, including the database, regardless of the ACL for the object. The DBADM privilege is the most powerful privilege in Oracle Rdb because it can override most privilege checks performed by Oracle Rdb. Users with the DBADM database privilege implicitly receive *all* privileges for all objects, except the SECURITY database privilege.
- Users with the SECURITY database privilege implicitly receive the Oracle Rdb SELECT, INSERT, UPDATE, and DELETE database privileges and the Oracle Rdb DBCTRL database and table privileges.

Digital UNIX

On Digital UNIX, the users root and dbsmgr can perform any operation on any named object, including the database, regardless of the ACL of the object. ♦

OpenVMS VAX OpenVMS Alpha

- Users with the OpenVMS SYSPRV privilege implicitly receive the same privileges as users with the DBADM database privilege.

  Users with the OpenVMS OPER privilege implicitly receive the SELECT, INSERT, UPDATE and DELETE database privileges.

  Users with the OpenVMS SECURITY privilege implicitly receive the same privileges as users with the SECURITY database privilege.

  Users with the OpenVMS BYPASS privilege implicitly receive *all* privileges except the Oracle Rdb DBADM and SECURITY database privileges and the DBCTRL database and table privileges.

  Users with the OpenVMS READALL privilege implicitly receive Oracle Rdb SELECT and SHOW database and table privileges. ♦

- You must execute the GRANT statement in a read/write transaction. If you issue this statement when there is no active transaction, Oracle Rdb starts a read/write transaction implicitly.
- You cannot execute the GRANT statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Maintenance* for more information on the RDB$SYSTEM storage area.
- You can deny users the right to create databases.

  OpenVMS VAX OpenVMS Alpha

  – On OpenVMS, you can use the RDBVMS$CREATE_DB logical name along with the RDBVMS$CREATE_DB rights identifier to deny users the right to create databases. See the *Oracle Rdb7 Guide to Database Design and Definition* for more information about the RDBVMS$CREATE_DB logical name. ♦

  **Caution**

  When you use the RDBVMS$CREATE_DB logical name, other installed third-party products will not be able to use Oracle Rdb to create Oracle Rdb databases. Therefore, you must deassign this logical name whenever users of such products need to create an Oracle Rdb database.

- You cannot GRANT privileges on stored procedures or stored functions.

For more information on protection for an Oracle Rdb database, see the chapter on defining privileges in the *Oracle Rdb7 Guide to Database Design and Definition*.

## Examples

Example 1: Redeclaring a database to make ACL changes take effect

This example illustrates that GRANT and REVOKE statements do not take effect until you attach to the database again.

```
SQL> -- Display the ACL for the EMPLOYEES table:
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
Protection on Table EMPLOYEES
    (IDENTIFIER=[sql,warring],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+ALTER+DROP)
SQL>
SQL> -- User warring, the owner of the database, denies
SQL> -- herself INSERT access to the EMPLOYEES table:
SQL> REVOKE INSERT ON TABLE EMPLOYEES FROM warring;
SQL> COMMIT;
SQL>
SQL> -- The SHOW PROTECTION statement displays the change
SQL> -- (INSERT is no longer part of the ACL entry
SQL> -- for warring):
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
Protection on Table EMPLOYEES
    (IDENTIFIER=[sql,warring],ACCESS=SELECT+UPDATE+DELETE+SHOW+CREATE+ALTER+
      DROP+DBCTRL+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+ALTER+DROP)
SQL>
SQL> -- But the change is not yet effective.
SQL> -- User warring can still store rows in the EMPLOYEES table:
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ('99999');
1 row inserted
SQL> SELECT EMPLOYEE_ID
cont>   FROM EMPLOYEES
cont>   WHERE EMPLOYEE_ID = '99999';
 EMPLOYEE_ID
 99999
1 row selected
SQL> ROLLBACK;
SQL>
SQL> -- To make the ACL change take effect, issue another ATTACH statement
SQL> -- to override the current declaration:
SQL> ATTACH 'FILENAME personnel';
This database context has already been declared.
Would you like to override this declaration (No)? Y
SQL>
SQL> -- Now warring cannot insert new rows into the EMPLOYEES table:
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ("99999");
%RDB-E-NO_PRIV, privilege denied by database facility
SQL>
SQL> -- A GRANT statement gives all privileges back to warring:
SQL> GRANT ALL ON TABLE EMPLOYEES TO warring;
SQL> COMMIT;
```

Example 2: Creating an ACL with an SQL command file

The following SQL command file creates an ACL for the default database by specifying the default alias RDB$DBHANDLE. It uses two general guidelines for ordering ACL entries:

- The less restrictive the user identifier, the lower on the list that ACL should go.
- The more powerful the privilege, the higher on the list that ACL should go.

Because SQL reads the list from top to bottom, you should place entries with more specific identifiers earlier, and those with more general ones later. For example, if you place the entry with the most general user identifier, [*,*], first in the list, all users match it, and Oracle Rdb grants or denies all the access rights specified there to all users.

Similarly, if you place the general entry [admin,*] before the specific entry [admin,ford], SQL matches user [admin,ford] with [admin,*] and denies the access rights INSERT, UPDATE, and DELETE, which user [admin,ford] needs.

In the following SQL command file, rights identifiers, such as PROGRAMMERS, are valid only on OpenVMS systems:

```
-- Database Administrator -- needs all privileges.
--
     GRANT ALL
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [group2,adams]
     POSITION 1;

-- Assistant -- needs to be able to use data definition statements.
--
     GRANT SELECT,CREATE,ALTER,DROP
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [group2,clark]
     POSITION 2;

-- Operator -- needs to be able to perform database maintenance tasks.
--
     GRANT SELECT, ALTER, DBADM
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [group2,lawrence]
     POSITION 3;
```

```
-- Security Administrator -- needs to specify and show security events
-- audited for a database and review the audit trail.
--
     GRANT SECURITY
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [group2,davis]
     POSITION 4;

-- Manager -- needs to be able to use all data manipulation statements.
--
     GRANT SELECT,INSERT,UPDATE,DELETE
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [admin,smith]
     POSITION 5;

-- Secretary -- needs to be able to read, write, and delete data.
-- No access to data definition or maintenance.
--
     GRANT SELECT,INSERT,UPDATE,DELETE
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [admin,ford]
     POSITION 6;

-- Programmers -- need to perform data definition and data manipulation
-- on some tables and constraints to test application programs.
--
     GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,ALTER,DROP,REFERENCES
     ON DATABASE ALIAS RDB$DBHANDLE
     TO PROGRAMMERS
     POSITION 7;

-- Clerks -- need to be able only to read data. No access to modify, erase,
-- store, data definition, or maintenance statements.
--
     GRANT SELECT
     ON DATABASE ALIAS RDB$DBHANDLE
     TO [admin,*]
     POSITION 8;

-- Deny access to all users not explicitly granted access to the database.
--
     REVOKE ALL
     ON DATABASE ALIAS RDB$DBHANDLE
     FROM PUBLIC
     POSITION 9;
```

Example 3: Granting column access and denying table access

You need the REFERENCES privilege to define constraints that affect a particular column. You need the UPDATE privilege to update data in a column. A user with the UPDATE privilege for a table automatically receives the UPDATE privilege for all columns in that table. To update a column, you must have the UPDATE privilege either for the column or for the table. However, a database administrator can restrict UPDATE privileges by defining them only for columns users should be able to update, and then removing the UPDATE privilege from the table entry. Because current salary is sensitive information, you might want to restrict the ability to update this amount.

The following example prevents user [admin,ford] from updating any column in the SALARY_HISTORY table except SALARY_START and SALARY_END. For instance, user [admin,ford] *cannot* update the SALARY_AMOUNT column.

```
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.SALARY_START
cont> TO [admin,ford];
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.SALARY_END
cont> TO [admin,ford];
SQL> --
SQL> REVOKE UPDATE ON TABLE SALARY_HISTORY FROM [admin,ford];
SQL> --
SQL> COMMIT;
SQL> --
SQL> SHOW PROTECTION ON TABLE SALARY_HISTORY;
Protection on Table SALARY_HISTORY
    (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
     ALTER+DROP+DBCTRL+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> --
SQL> SHOW PROTECTION ON COLUMN SALARY_HISTORY.SALARY_START;
Protection on Column SALARY_HISTORY.SALARY_START
    (IDENTIFIER=[admin,ford],ACCESS=UPDATE)
    (IDENTIFIER=[*,*],ACCESS=NONE)
```

## GRANT Statement, ANSI/ISO-Style

Creates or adds ANSI/ISO-style privileges to an entry of the Oracle Rdb access privilege set for a database, table, view, column, module, or external routine. At database creation time, you specify whether the database protection mechanism will be ANSI/ISO-style or ACL-style. For more information on creating or changing the style of privileges associated with a database, see the CREATE DATABASE Statement.

Each entry in an ANSI/ISO-style access privilege set consists of an identifier and a list of privileges assigned to the identifier.

- Each identifier specifies a user or PUBLIC access.
- The set of privileges specifies what operations that user can perform on the database, table, column, module, or external routine.

ANSI/ISO-style privileges:

- Grant access to the creator when an object is created. Because only the creator is granted access to the newly created object, additional access must be granted explicitly.
- Support only the PUBLIC identifier as a wildcard.
- Support only user identifiers that translate to an OpenVMS user identification code (UIC) or a Digital UNIX user identifier (UID).

For ANSI/ISO-style databases, a user's privileges are a combination of all privilege sets that apply to that user. The access privilege set is not order-dependent. The user matches the entry in the access privilege set; receives whatever privileges have been granted for the database, table, column, module, or external routine; and receives the privileges defined for PUBLIC. A user without an entry in the access privilege set receives only the privileges defined for PUBLIC, which always has an entry in the access privilege set even if PUBLIC has no access to the database, table, column, module, or external routine.

To remove privileges from or entirely delete an entry to the Oracle Rdb access privilege set for a database, table, column, module, or external routine, see the REVOKE Statement, ANSI/ISO-Style.

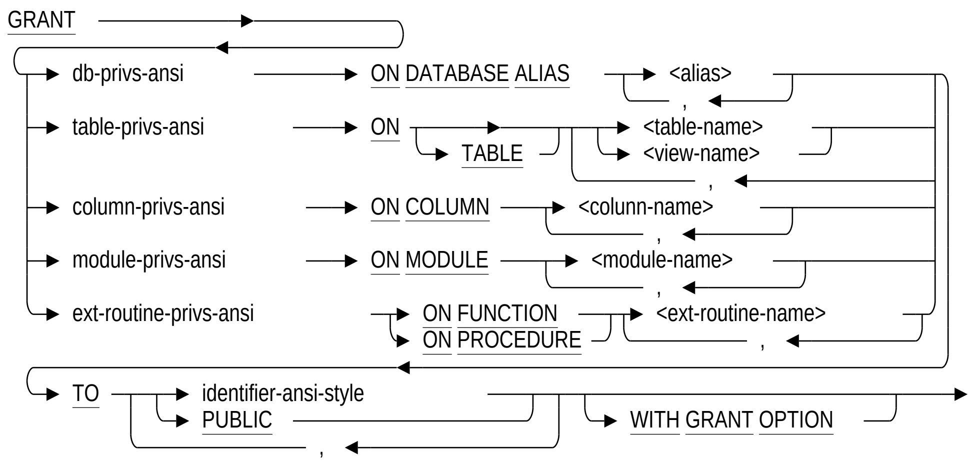
## GRANT Statement, ANSI/ISO-Style

### Environment

You can use the GRANT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



db-privs-ansi =

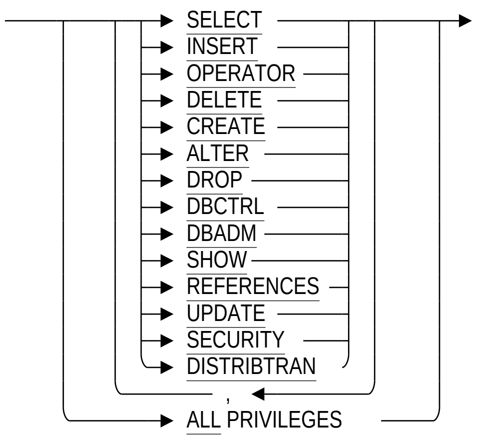
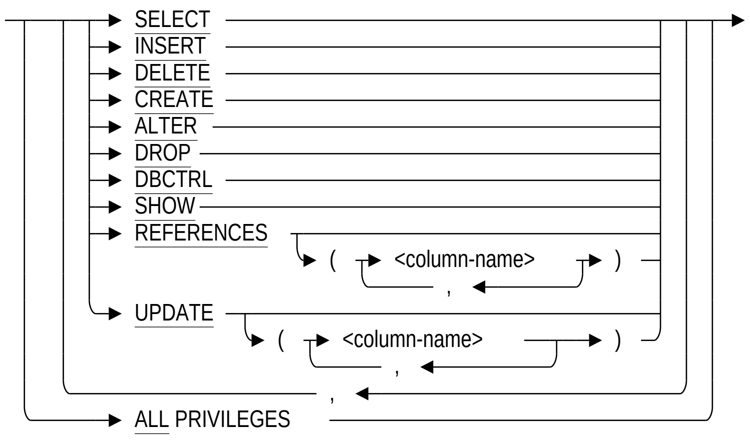
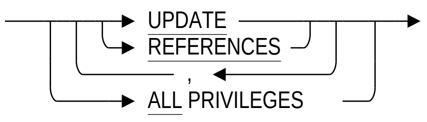


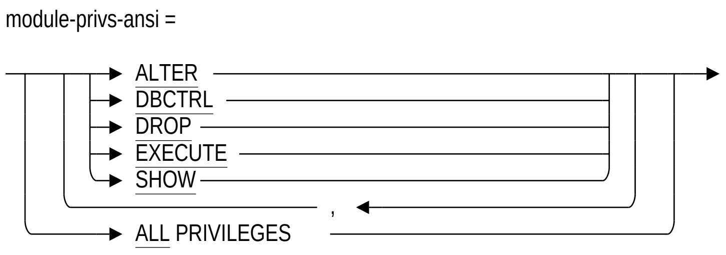
table-privs-ansi =

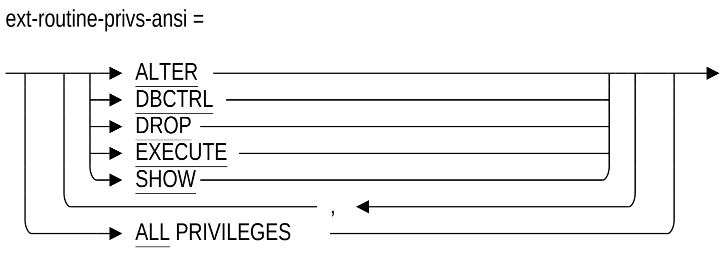


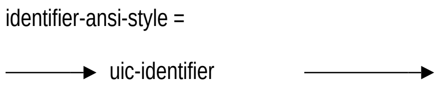
column-privs-ansi =



## GRANT Statement, ANSI/ISO-Style







## Arguments

**ON DATABASE ALIAS alias**
**ON TABLE table-name**
**ON TABLE view-name**
**ON COLUMN column-name**
**ON MODULE module-name**
**ON FUNCTION ext-routine-name**
**ON PROCEDURE ext-routine-name**
Specifies whether the GRANT statement applies to databases, tables, columns, views, modules, or external routines. You can specify a list of names for any form of the ON clause. You must qualify a column name with at least the associated table name.

**db-privs-ansi**
**table-privs-ansi**
**column-privs-ansi**
**module-privs-ansi**
**ext-routine-privs-ansi**
Specifies the set of privileges you want to add to an existing access privilege set entry or create in a new one. The operations permitted by a given privilege keyword differ, depending on whether you granted it for a database, table, column, module, or external routine. Table 7–1 (in the GRANT Statement) lists the privilege keywords and their meanings for databases, tables, and columns.

The REFERENCES privilege lets you define a constraint for a database with ANSI/ISO-style privileges.

**ALL PRIVILEGES**
Specifies that SQL should grant all privileges to the specified users.

**TO identifier-ansi-style**
Specifies the identifiers for the new or modified access privilege set entry. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

In ANSI/ISO-style databases, you are allowed to specify only single-user user identifiers; no general or system identifiers are allowed. Access privilege set entries identify only those users who are common to all groups defined by the individual identifiers. Users who do not match all identifiers are not controlled by that entry. ANSI/ISO-style access privilege sets support only user identifiers.

**user-identifier**
Specifies a user identifier that uniquely identifies each user on the system.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the user identifier consists of the standard OpenVMS user identification code (UIC), a group name and a member name (user name). The group name is optional. The user identifier can be in either numeric or alphanumeric format. The following are all valid user identifiers that could identify the same user:

K_JONES
[SYSTEM3, K_JONES]
[341,311] ♦

Digital UNIX

On Digital UNIX, a user identifier consists of the name of a group and the standard Digital UNIX user identifier (UID). The group name is optional. The user identifier must be in alphanumeric form. Note that on Digital UNIX, users can belong to more than one group. The following are valid user identifiers that could identify the same user:

k_jones
[system3, k_jones] ♦

In ANSI/ISO-style user identifiers, the only wildcard allowed is in the public identifier [*,*].

When Oracle Rdb creates an ANSI/ISO-style database, the creator of the database gets all privileges, and the PUBLIC entry gets no privileges.

In an ANSI/ISO-style database, you cannot use multiple user identifiers for any entry except PUBLIC.

For more information about identifiers, see your operating system documentation.

**WITH GRANT OPTION**
Allows the user who has been granted a privilege the option of granting that privilege to other users.

The WITH GRANT OPTION clause specifies that the grantees in the TO clause may grant the privileges in the privilege list to other users for as long as they have the privileges. When the privilege is revoked from the grantee who received the privileges with the WITH GRANT OPTION clause, the privileges also are revoked from all the users who received the privileges from that grantee (unless these users have received the privilege from yet another user who still has the privilege).

## Usage Notes

- For information on how to set up an ANSI/ISO-style database protection mechanism, see the CREATE DATABASE Statement.
- Additions and changes to access privilege sets do not take effect until you declare the database again, even though those changes are displayed by the SHOW PROTECTION and SHOW PRIVILEGES statements. Additions and changes to access privilege sets do not take effect for other users until they declare the database again.

- You must declare all databases that you refer to in a GRANT (ANSI-style) statement. If you use the default database declaration, you must use the alias RDB$DBHANDLE to work with database access privilege sets.
- You can use the GRANT (ANSI/ISO-style) statement to modify existing access privilege set entries or create new ones.

  To modify an existing access privilege set entry, specify the same identifier in the TO clause as is in the existing entry.

  To create a new access privilege set entry, specify an identifier that is not already part of an entry.
- Digital UNIX

  On Digital UNIX, if you do not specify a group, SQL displays the group specified in the operating system's password file, as shown in the following example:

```
SQL>  GRANT ALL ON DATABASE ALIAS RDB$DBHANDLE TO janasik;
SQL> SHOW PROTECTION ON DATABASE RDB$DBHANDLE
Protection on Alias RDB$DBHANDLE

(IDENTIFIER=[users,janasik],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[users,heleng],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+
      CREATE+ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[*,*],ACCESS=NONE)                    ♦
```

- You cannot execute the GRANT statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Maintenance* for more information on the RDB$SYSTEM storage area.
- Users with the DBADM database privilege can perform any data definition or data manipulation operation on any named object, including the database, regardless of the ACL for the object. The DBADM privilege is the most powerful privilege in Oracle Rdb because it can override most privilege checks performed by Oracle Rdb. Users with the DBADM database privilege implicitly receive *all* privileges for all objects, except the SECURITY database privilege.
- OpenVMS VAX OpenVMS Alpha

  Users with the OpenVMS SYSPRV privilege implicitly receive the same privileges as users with the DBADM database privilege.

  Similarly, users with the OpenVMS READALL privilege are implicitly granted SELECT and SHOW privileges to the database.♦

Digital UNIX

- On Digital UNIX, the users root and dbsmgr can perform any operation on any named object, including the database, regardless of the ACL of the object.

  For information about other Oracle Rdb and operating system privileges that have some override capabilities, see the GRANT Statement. ♦
- You must execute the GRANT (ANSI/ISO-style) statement in a read/write transaction. If you issue this statement when there is no active transaction, Oracle Rdb starts a read/write transaction implicitly.
- Privileges on a column are determined by the privileges defined on the table combined with those specified for the specific column access privilege set.
- A user with UPDATE or REFERENCES privilege on the table automatically receives the same privileges on all columns in the table. With UPDATE and REFERENCES privileges, you must have the privilege for either the column or the table to update a column. However, you can restrict UPDATE and REFERENCES privileges by defining them only on specific columns you want users to be able to update or define constraints for, and thus remove the privilege from the table entry.

  You can modify the data in a column only if you have the UPDATE privilege for the column and the SELECT privilege for the database.
- The REFERENCES privilege lets you define a constraint.
- You cannot GRANT privileges on stored procedures or stored functions.

For more information on protection for an Oracle Rdb database, see the chapter on defining privileges in the *Oracle Rdb7 Guide to Database Design and Definition*.

## Examples

Example 1: Reattaching to a database to make access privilege set changes take effect

This example illustrates that GRANT and REVOKE statements do not take effect until the user granting privileges commits the changes, and the user to whom privileges are granted attaches to the database again.

```
SQL> GRANT SELECT ON DATABASE RDB$DBHANDLE TO rdml_doc;
SQL> COMMIT;
SQL> -- To make the access privilege set change take effect,
SQL> -- user granted privileges issues another ATTACH statement
SQL> -- to override the current declaration:
SQL> ATTACH 'FILENAME ansi_test';
This alias  has already been declared.
Would you like to override this declaration (No)? Y
```

Example 2: Using PUBLIC as a wildcard

```
SQL> SHOW PROTECTION ON DATABASE RDB$DBHANDLE;
Protection on Alias RDB$DBHANDLE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
[rdml_doc]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
SQL> GRANT SELECT ON DATABASE RDB$DBHANDLE TO PUBLIC;
SQL> -- Then grant the privilege on the appropriate tables
SQL> GRANT SELECT ON EMPLOYEES TO PUBLIC;
SQL> COMMIT;
SQL> ATTACH 'FILENAME ansi_test';
This alias has already been declared.
Would you like to override this declaration (No)? Y
SQL> SHOW PROTECTION ON DATABASE RDB$DBHANDLE;
Protection on Alias RDB$DBHANDLE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
SQL> SHOW PROTECTION ON EMPLOYEES;
Protection on Table EMPLOYEES
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
```

Example 3: Granting a privilege with the WITH GRANT OPTION clause

This example shows how the grantee can grant the privilege until the clause is revoked from the grantor's privilege.

```
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
[sql,warring]:
  With Grant Option:        SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,
                            DROP,DBCTRL,,DBADM,REFERENCES
  Without Grant Option:     SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,
                            DROP,DBCTRL,DBADM,REFERENCES
[rdb,rdb_doc]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
SQL>
SQL> GRANT DELETE ON EMPLOYEES TO [rdb,rdb_doc] WITH GRANT OPTION;
SQL> COMMIT;
SQL> ATTACH 'FILENAME ansi_test';
This alias has already been declared.
Would you like to override this declaration (No)? Y
SQL> SHOW PROTECTION ON EMPLOYEES;
Protection on Table EMPLOYEES
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
[rdb, rdb_doc]:
  With Grant Option:        DELETE
  Without Grant Option:     NONE
SQL>
SQL> -- Show the list of users who will lose their DELETE privilege if
SQL> -- the privilege is taken away from warring.
SQL> SHOW USERS WITH DELETE ON EMPLOYEES FROM warring;
Users granted privileges on Table EMPLOYEES by [sql,warring]
[sql,poor]
[rdb,rdb_doc]
SQL>
SQL> -- Check if anyone on the list has given DELETE privilege to anyone else:
SQL> SHOW USERS GRANTING DELETE ON EMPLOYEES TO PUBLIC;
Users granting privileges on Table EMPLOYEES to [*,*]
No users found
```

Example 4: Granting column privileges

This example shows how to grant privileges on a specific column.

```
SQL> -- Give UPDATE privilege on one column:
SQL> GRANT UPDATE (CANDIDATE_STATUS) ON TABLE CANDIDATES TO CARPENTER;
SQL> --
SQL> -- Give UPDATE privilege on another column:
SQL> GRANT UPDATE ON COLUMN EMPLOYEES.EMPLOYEE_ID TO rdb_doc;
SQL> COMMIT;
SQL> SHOW PROTECTION ON COLUMN EMPLOYEES.EMPLOYEE_ID;
[rdb,rdb_doc]:
  With Grant Option:        NONE
  Without Grant Option:     UPDATE
SQL> User carpenter tries to update a table for which he does not
SQL> -- have UPDATE privilege on the FIRST_NAME column.
SQL> ATTACH 'FILENAME ansi_test';
This alias has already been declared.
Would you like to override this declaration (No)? Y
SQL> UPDATE CANDIDATES SET CANDIDATE_STATUS = "Hired"
cont>  WHERE LAST_NAME = "Wilson";
SQL> UPDATE CANDIDATES SET FIRST_NAME = "Felix"
cont>  WHERE LAST_NAME = "Wilson";
%RDB-E-NO_META_UPDATE, metadata update failed
-RDB-E-NO_PRIV, privilege denied by database facility
```

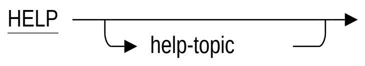
## HELP Statement

Gives you access to assistance on all SQL statements, components, and concepts.

### Environment

You can issue the HELP statement only in interactive SQL.

### Format



### Arguments

**topic**
The SQL statement or concept on which you need help.

### Usage Notes

- When you type HELP:
  - A menu of topics on which assistance is available replaces the SQL prompt (SQL>).
  - After the menu scrolls by, the cursor remains at a "Topic?" prompt. Typing any of the menu items yields assistance on that topic. Many of the topics have further levels of assistance, indicated by a "Subtopic?" prompt.
  - To move back to the next higher level, press the Return key. For example, pressing the Return key at the "Subtopic?" prompt brings you to the "Topic?" prompt, and pressing the Return key again returns you to the SQL prompt.
  - To see the list of additional topics at any level, type a question mark (?) and press the Return key.
  - To leave Help, enter Ctrl/Z or at the "Topic?" prompt, press the Return key.

- Most Help entries in SQL have a similar structure. The main screen shows a brief description of the topic and, if you requested help on a statement, a syntax diagram. In many cases, this screen gives you all the information you need to execute the statement.

  The main screen also displays a list of "Additional information available." This list usually includes these additional entries:

  – More: A more detailed description of the topic.

  – Arguments: Subtopics describing the arguments.

## Example

Example 1: Obtaining online Help in SQL

```
SQL> HELP SELECT
```

## IF Control Statement

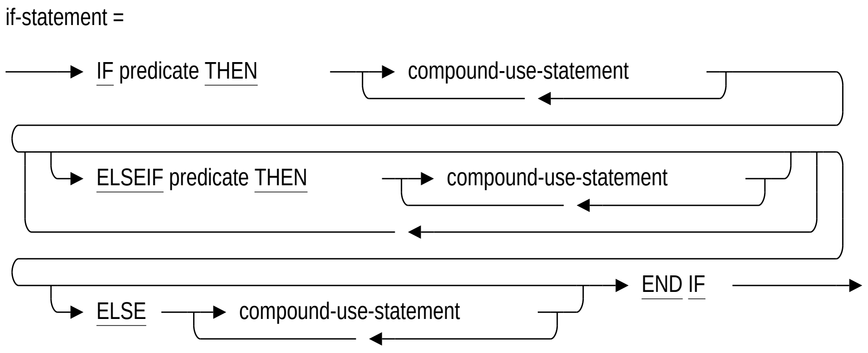
Executes one or more SQL statements conditionally. It then continues processing by executing any SQL statement that immediately follows the block.

### Environment

You can use the IF control statement in a compound statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**IF predicate THEN compound-use-statement**
Executes one or more SQL statements in an IF . . . END IF block only when the value of an IF predicate evaluates to TRUE. A **predicate**, also called a conditional expression, specifies a condition that SQL evaluates to TRUE, FALSE, or UNKNOWN. If the predicate evaluates to TRUE, SQL executes the statement in the THEN clause. If the predicate does not evaluate to TRUE, SQL evaluates the predicate in any ELSEIF clauses. If the IF statement contains no ELSEIF clauses, SQL executes any statements in the ELSE clause.

**predicate**
See Section 2.7 for more information on predicates.

**compound-use-statement**
See the Compound Statement for a description of the SQL statements that are valid in a compound statement.

**ELSEIF predicate THEN compound-use-statement**
If the ELSEIF predicate evaluates to TRUE, SQL executes the SQL statements in the THEN clause. If the ELSEIF predicate does not evaluate to TRUE, SQL evaluates the predicates in any subsequent ELSEIF clauses.

If an ELSE clause follows the ELSEIF clause, SQL executes the SQL statements associated with the ELSE clause.

**ELSE compound-use-statement**
Executes one or more SQL statements associated with the ELSE clause but only when the value of the IF and ELSEIF predicates evaluate to FALSE or UNKNOWN.

**END IF**
Marks the end of an IF statement. Every IF statement must end with the END IF clause.

## Usage Notes

- As with all compound statements, you can nest IF statements.
- Using the ELSEIF clause instead of a nested IF statement can make your code easier to read. While both methods produce the same results, using nested IF statements can obscure logic flow.
- When SQL drops out of the IF . . . END IF block, it then continues processing by executing any SQL statement that immediately follows the block.
- The testing of predicates proceeds from the IF clause to each of the ELSEIF clauses in the order in which they appear. The statements of the first IF or ELSEIF clause that evaluates to TRUE are executed. The statements of the ELSE clause are executed if none of these is TRUE. Under no circumstance is more than one type of an IF statement executed.

## Examples

Example 1: Using an IF control statement

```
IF (SELECT COUNT (*) FROM STUDENTS
           WHERE CLASS = :CLASS_NUM)
     > 30

   THEN
      SET :MSG = ’Class is too large.’;
   ELSE
      SET :MSG = ’Class size is O.K.’;

END IF;
```

## IMPORT Statement

Creates an Oracle Rdb database from an interchange .rbr file.

You use the IMPORT statement with the EXPORT statement to make changes to Oracle Rdb databases that cannot be made any other way. The EXPORT statement unloads a database to an .rbr file. The IMPORT statement re-creates the database with changes that are both allowed and not allowed through ALTER statements. The IMPORT statement lets you:

- Convert from a single-file to a multifile database, and vice versa.
- Change database root file parameters that you cannot change with the ALTER DATABASE statement:
  - COLLATING SEQUENCE
  - BUFFER SIZE
  - SEGMENTED STRING STORAGE AREA
  - PROTECTION IS ANSI/ACL
- Change storage area parameters that you cannot change with the ALTER DATABASE statement:
  - PAGE SIZE
  - PAGE FORMAT
  - THRESHOLDS
  - INTERVAL
  - SNAPSHOT FILENAME
- Reload tables with existing rows to take advantage of newly created hashed indexes.
- Reload tables to take advantage of new or changed storage maps.
- Move a database to another directory or disk structure. However, if moving a database is the only change you need to make, it is more efficient to use the RMU Backup and RMU Restore commands.
- Create an empty target database that uses the same data definitions as a source database by copying the metadata, but not the data, to the target.

If you use the NO DATA option, the IMPORT statement creates an Oracle Rdb database whose metadata is identical to that found in the source database used by the EXPORT statement, but the duplicate database contains no data. The NO DATA option is not compatible with the repository databases. See the description in the Arguments section under the NO DATA option.

OpenVMS VAX OpenVMS Alpha SQL uses the following rules to determine character set information when you export and import a database:

- When you export a V4.2 or higher database and import it to a V4.1 or lower version, SQL ignores information about the database character set in the database. Lower versions of Oracle Rdb rely on the character set logical name (RDB$CHARACTER_SET) to specify the character set. The logical name is deprecated and will not be supported in a future release.
- When you import a database to V4.2 or higher from an export file created by V4.1 or lower, SQL uses the logical name RDB$CHARACTER_SET to determine the database default character set and national character set.

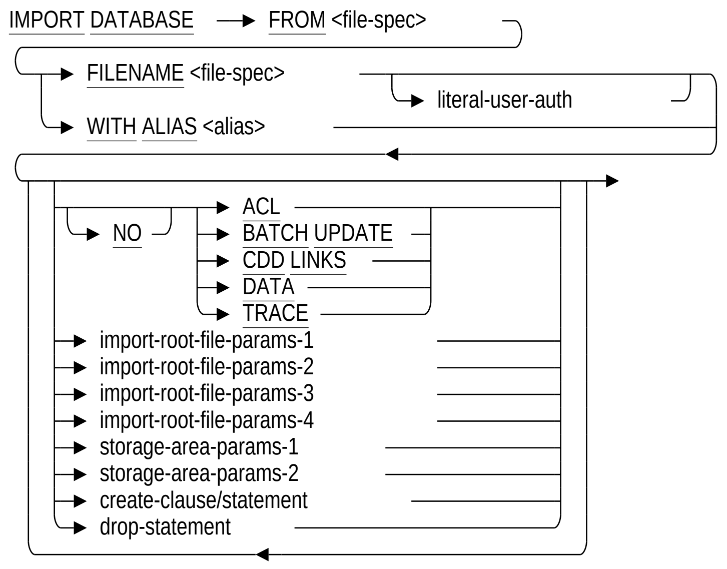
  If the logical name is not defined, SQL uses DEC_MCS as the database default character set.
- When you import a database to V4.2 or higher from an export file created by V4.2 or lower, SQL ignores the values in the logical name and uses the values stored in the database.
- If the logical name is defined, but the equivalence name is not a valid name, SQL returns an error and the conversion fails. See Table E–2 for a list of the valid equivalence names.

♦

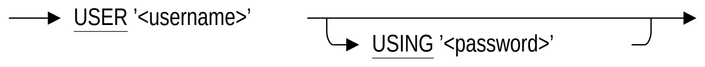
## Environment

You can use the IMPORT statement in interactive SQL only.

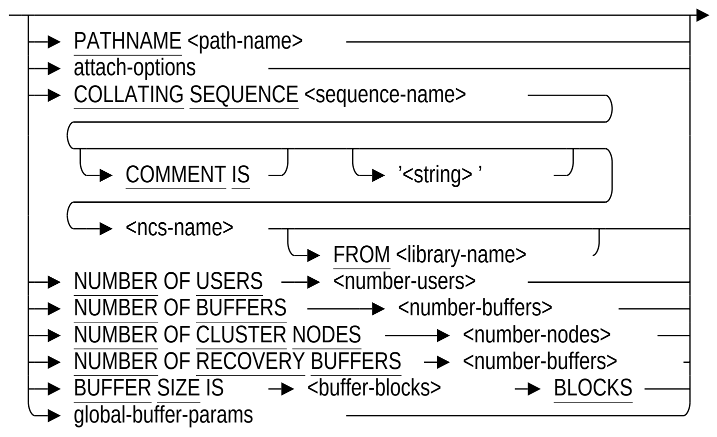
## Format



literal-user-auth =

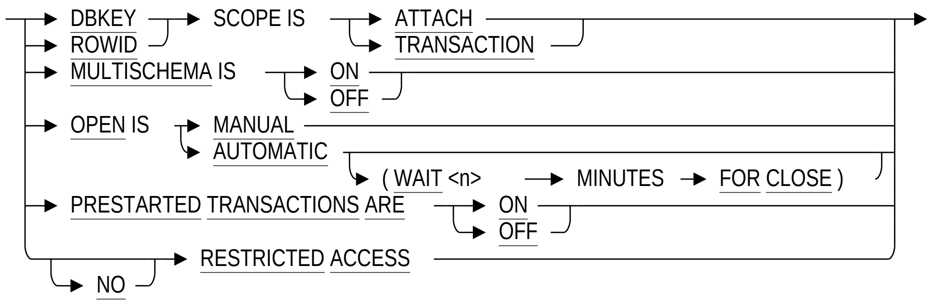


import-root-file-params-1 =

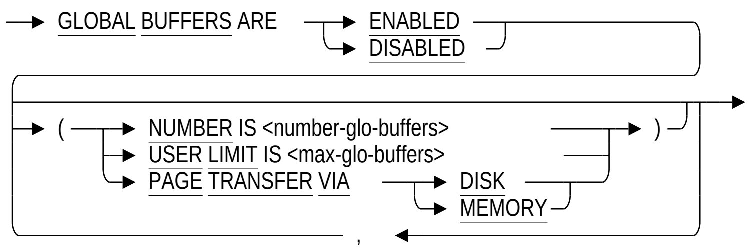


## IMPORT Statement

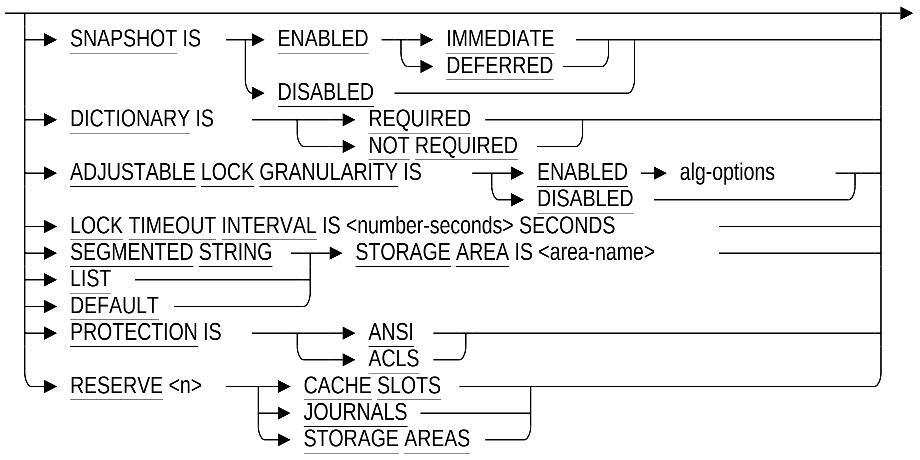
attach-options =



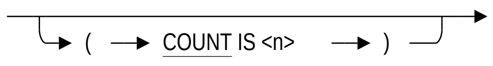
global-buffer-params=



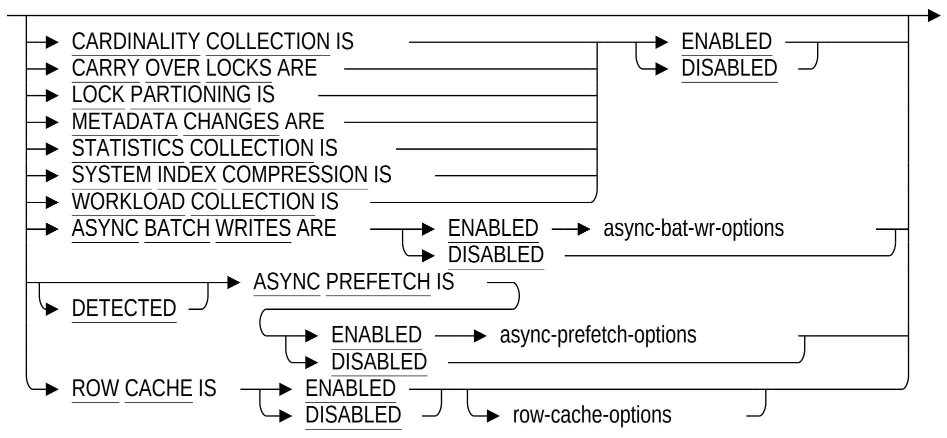
import-root-file-params-2 =



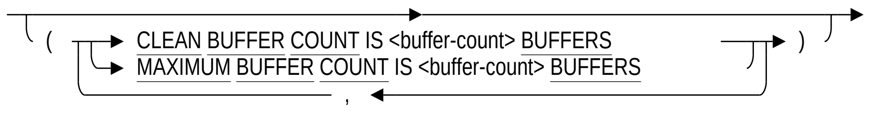
alg-options =



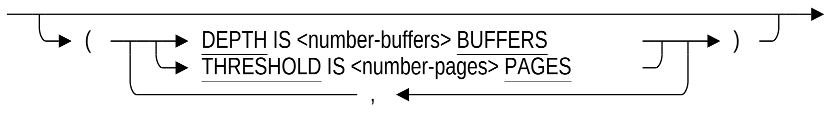
import-root-file-params-3 =



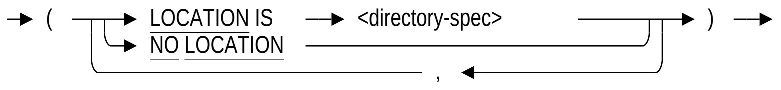
asynch-bat-wr-options =


(
CLEAN BUFFER COUNT IS <buffer-count> BUFFERS
MAXIMUM BUFFER COUNT IS <buffer-count> BUFFERS
,
)

async-prefetch-options =

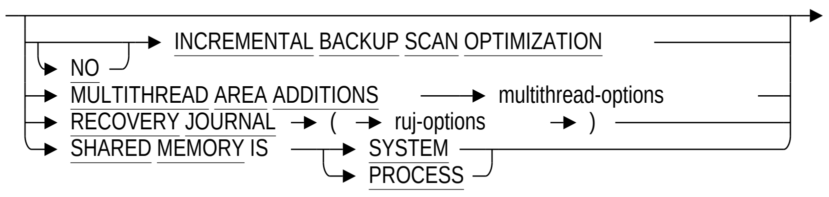

(
DEPTH IS <number-buffers> BUFFERS
THRESHOLD IS <number-pages> PAGES
,
)

row-cache-options =

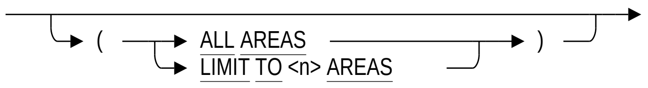


## IMPORT Statement

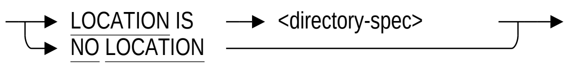
import-root-file-params-4 =



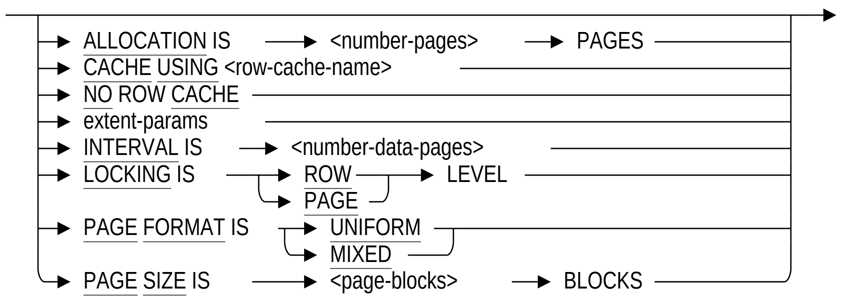
multithread-options =



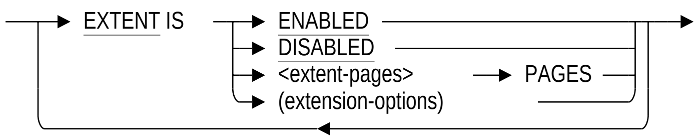
ruj-options =



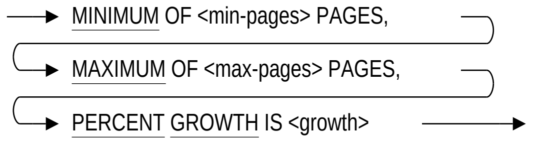
storage-area-params-1 =



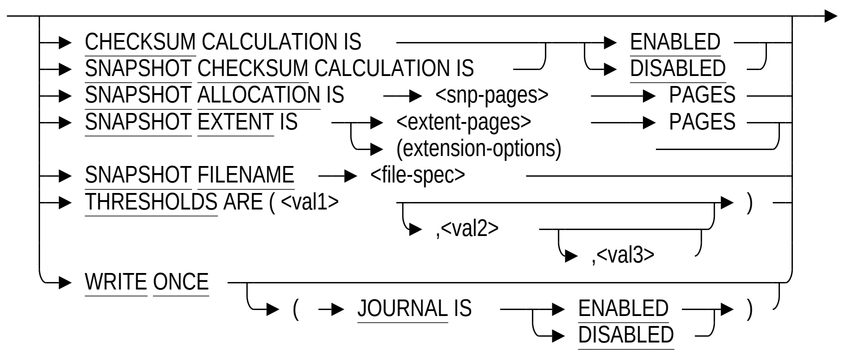
extent-params =



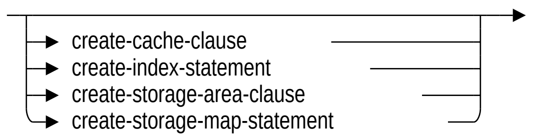
extension-options =


MINIMUM OF <min-pages> PAGES,
MAXIMUM OF <max-pages> PAGES,
PERCENT GROWTH IS <growth>

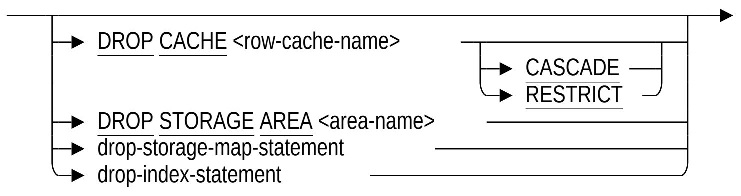
storage-area-params-2 =



create-clause/statement =



drop-statement =


DROP CACHE <row-cache-name>
CASCADE
RESTRICT
DROP STORAGE AREA <area-name>
drop-storage-map-statement
drop-index-statement

## Arguments

**FROM file-spec**
Names the interchange .rbr file that the IMPORT statement uses as a source to create a new database.

**FILENAME file-spec**
Specifies the file associated with the database.

If you omit the FILENAME argument, the file specification takes the following defaults:

- Device: the current device for the process (on OpenVMS only)
- Directory: the current directory for the process
- File name: the alias (if you omit the FILENAME argument, you must specify the WITH ALIAS clause)

OpenVMS VAX OpenVMS Alpha Use either a full file specification or a partial file specification. You can use a logical name for all or part of a file specification. ♦

If you use a simple file name, SQL creates the database in the current default directory. Because the IMPORT statement may create more than one file with different file extensions, do not specify a file extension with the file specification.

The files created using the file specification in the FILENAME clause depend on whether you create a multifile or single-file database.

- In multifile IMPORT statements, SQL uses the file specification to create up to three files:
  - A database root file with a file extension of .rdb
  - A storage area file, with a file extension of .rda, for the main storage area, RDB$SYSTEM (unless the IMPORT statement contains a CREATE STORAGE AREA RDB$SYSTEM statement, which overrides this file specification)
  - A snapshot file, with a file extension of .snp, for the main storage area, RDB$SYSTEM (unless the IMPORT statement contains a CREATE STORAGE AREA RDB$SYSTEM statement, which overrides this file specification)

  A multifile IMPORT statement is one that includes CREATE STORAGE AREA statements or uses an .rbr file from a multifile database without explicitly deleting all the storage areas.

- In single-file IMPORT statements, SQL uses the file specification to create two files:
  - A combined database root and storage area file with a file extension of .rdb
  - A snapshot file with a file extension of .snp (unless overridden by a SNAPSHOT FILENAME clause in the storage area parameters)

  A single-file IMPORT statement is one that omits CREATE STORAGE AREA statements and uses an .rbr file from a single-file database or explicitly deletes all the storage areas in a multifile database.

**literal-user-auth**
Specifies the user name and password for access to databases, particularly remote database.

This literal lets you explicitly provide user name and password information in the IMPORT statement.

**USER ’username’**
Defines a character string literal that specifies the operating system user name that the database system uses for privilege checking.

**USING ’password’**
Defines a character string literal that specifies the user’s password for the user name specified in the USER clause.

**WITH ALIAS alias**
Specifies the alias for the implicit database attach executed by the IMPORT statement. An alias is a name for a particular attachment to a database.

You must specify an alias or a file name. If you omit the WITH ALIAS clause, the default alias for the database created by the IMPORT statement is RDB$DBHANDLE. If you omit the FILENAME argument, the IMPORT statement also uses the alias as the file name for the database root file and creates the root file in the current default directory. If you omit WITH ALIAS, you must specify the FILENAME argument.

**ACL**
**NO ACL**
Specifies that the IMPORT statement uses the access control lists from the original database when it creates the new database. The ACL option is the default. If you are using the IMPORT statement to restructure a database, you typically want to use the ACL option and preserve the access control lists.

The NO ACL option overrides the ACLs from the original database and uses the database system default ACLs. Specify NO ACL if you are using the IMPORT statement to rebuild a database on a different system, or to convert to an Oracle Rdb database from another supported database system. The NO ACL option makes you the owner of the new database and creates default access control lists.

**BATCH UPDATE**
**NO BATCH UPDATE**
Specifies whether the IMPORT statement stores user data and indexes in a single batch-update transaction (BATCH UPDATE) or in separate read/write transactions for each table (NO BATCH UPDATE). The NO BATCH UPDATE option is the default.

A batch-update transaction is faster but does not perform recovery-unit journaling, which means you cannot recover the database in the event of a failure during the IMPORT operation. With the NO BATCH UPDATE option, you can recover the database.

For more information about batch-update transactions, see the SET TRANSACTION Statement.

**CDD LINKS**
**NO CDD LINKS**
OpenVMS VAX OpenVMS Alpha

Determines whether the IMPORT statement tries to reestablish links between database definitions originally based on repository definitions (domains and tables created with the FROM path name clause) and their sources in the repository.

The default depends on whether or not the IMPORT statement specifies the PATHNAME option. If the IMPORT statement does specify PATHNAME, the default is CDD LINKS; if it does not specify PATHNAME, the default is NO CDD LINKS.

The CDD LINKS option specifies that the IMPORT statement tries to reestablish repository links even if you do not specify the PATHNAME option. If you specify CDD LINKS and the database repository definition on which a database definition was based does not exist, the IMPORT statement generates a warning message.

The NO CDD LINKS option specifies that the IMPORT statement does not establish data repository links even if you specify the PATHNAME option. Specify NO CDD LINKS if you are using the IMPORT statement to rebuild a database on a different system or to convert to an Oracle Rdb database from another supported database system.

The CDD LINKS and NO CDD LINKS clauses are available only on OpenVMS platforms. ♦

**DATA**
**NO DATA**
Specifies whether the database created by the IMPORT statement includes the data and metadata contained in the source database, or the metadata only. DATA is the default.

When you specify the NO DATA option, you import the metadata that defines a database from an .rbr file and exclude the data. Duplicating the metadata of a database while excluding the data offers the following benefits:

- You can use established, tested metadata to create a database to store new data. Standardized metadata can be created once but used in multiple databases.
- You can use the duplicated metadata to test the database structure. You can experiment with storage areas and storage maps, and by entering sample data, you can test other aspects of database structure.
- If a database needs testing by someone outside of your group, you can submit the database metadata without exposing any sensitive data. Also, if the database is very large, you need not submit multiple reels of tape to the tester.

**Note**

OpenVMS VAX OpenVMS Alpha

The NO DATA option is not compatible with repository databases (CDD$DATABASE.RDB). An .rbr file, created by an EXPORT statement with the DATA option (the default) and generated from a CDD$DATABASE.RDB file, cannot be used with the NO DATA option for the IMPORT statement. SQL issues an error message stating that the NO DATA option is not valid for repository databases. ♦

**TRACE**
**NO TRACE**
Specifies whether usage statistics are logged by the IMPORT statement. The NO TRACE option is the default.

Some actions taken by the IMPORT statement can consume significant amounts of I/O resources and CPU time. These actions include the following operations:

- Loading data
- Defining indexes
- Defining constraints

When you specify the TRACE option with the IMPORT statement, SQL writes a message to your terminal screen when each operation begins, and writes a summary of DIO (direct input/output operations), CPU, and PAGE FAULT statistics when the operation completes. When the IMPORT statement finishes execution, a summary of all DIO, CPU, and PAGE FAULT statistics is displayed. The display also includes information on access to the .rbr file, database creation, and loading of data. For more information about these statistics, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

**import-root-file-params-1**
**import-root-file-params-2**
**import-root-file-params-3**
**import-root-file-params-4**
Parameters that control the characteristics of the database root file associated with the database, or characteristics stored in the database root file that apply to the entire database.

The PATHNAME, PRESTARTED TRANSACTIONS, and PROTECTION IS parameter descriptions follow. For more information on the other "import-root-file-params-1", "import-root-file-params-2", "import-root-file-params-3", and "import-root-file-params-4", see the descriptions of "root-file-params-1", "root-file-params-2", "root-file-params-3", and "root-file-params-4" in the CREATE DATABASE Statement.

**PATHNAME path-name**
OpenVMS VAX OpenVMS Alpha
Specifies the path name for the repository where the database definition is stored.

Specify either a:

- Full repository path name, such as CDD$TOP.SQL.DEPT3
- Relative repository path name, such as DEPT3
- Logical name that refers to a full or relative repository path name

If you use a relative path name, CDD$DEFAULT must be defined as all of the path name segments preceding the relative path name.

SQL does not update the repository unless you specify the PATHNAME clause. If you omit the PATHNAME clause, the database definition is not stored in the repository.

The PATHNAME clause is available only on OpenVMS platforms. ♦

**PRESTARTED TRANSACTIONS ARE ON**
**PRESTARTED TRANSACTIONS ARE OFF**

Specifies whether Oracle Rdb enables or disables prestarted transactions. If you use the PRESTARTED TRANSACTIONS ARE OFF clause, Oracle Rdb uses additional I/O because each SET TRANSACTION statement must reserve a transaction sequence number (TSN).

Use the PRESTARTED TRANSACTIONS ARE OFF clause only if your application uses a server process that is attached to the database for long periods of time and causes the snapshot file to grow excessively.

For most applications, Oracle Rdb recommends that you enable prestarted transactions. The default is PRESTARTED TRANSACTIONS ARE ON. If you use the PRESTARTED TRANSACTIONS ARE ON clause or do not specify the PRESTARTED TRANSACTIONS clause, the COMMIT or ROLLBACK statement for the previous read/write transaction automatically reserves the TSN for the next transaction and reduces I/O.

The PRESTARTED TRANSACTIONS clause refers only to the database attach that is performed as part of the IMPORT statement. The clause does not set permanent database attributes.

You can define the RDMS$BIND_PRESTART_TXN logical name or the RDB_BIND_PRESTART_TXN configuration parameter to define the default setting for prestarted transactions outside of an application. The PRESTARTED TRANSACTION clause overrides this logical name or configuration parameter. For more information, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

**PROTECTION IS ANSI**
**PROTECTION IS ACLS**

By default, the IMPORT statement retains the protection style of the database that was exported. However, if you specify PROTECTION IS ANSI or PROTECTION IS ACLS, then the IMPORT statement creates a database with that protection type. If the protection of the database created is different from the protection of the database that was exported, then no protection records are imported and you will receive default protections.

**storage-area-params**
Specifies parameters that control the characteristics of database storage area files. You can specify most storage area parameters for either single-file or multifile databases, but the effect of the clauses differs.

- For single-file databases, the storage area parameters specify the characteristics for the single storage area in the database.
- For multifile databases, the storage area parameters specify a set of default values for any storage areas created by the IMPORT statement that do not specify their own values for the same parameters. The attributes of a storage area are supplied by the interchange file unless redefined by the IMPORT statement. The default values apply to the storage area named in CREATE STORAGE AREA database elements.

  For details about storage area parameters, see the CREATE STORAGE AREA Clause.

**Note**

The CREATE STORAGE AREA clauses can override these default values. The default values do not apply to any storage areas created later with the ALTER DATABASE statement.

**create-cache-clause**
See the CREATE CACHE Clause for a complete description.

**create-index-statement**
See the CREATE INDEX Statement for a complete description.

**create-storage-area-clause**
See the CREATE STORAGE AREA Clause for a complete description.

**create-storage-map-statement**
See the CREATE STORAGE MAP Statement for a complete description.

**DROP CACHE row-cache-name CASCADE**
**DROP CACHE row-cache-name RESTRICT**
Deletes the specified row cache area from the database.

If the mode is RESTRICT, an exception is raised if the row cache area is assigned to a storage area.

If the mode is CASCADE, the row cache area is removed from all referencing storage areas.

The default is RESTRICT if no mode is specified.

**DROP STORAGE AREA area-name**
Deletes the specified storage area definition and the associated storage area and snapshot files. You can use the DROP STORAGE AREA clause only on multifile databases.

To protect against accidental data deletion, the IMPORT statement fails if you specify a DROP STORAGE AREA clause that names a storage area referred to in any storage map. You must first use the ALTER DATABASE . . . ALTER STORAGE MAP statement to move the data to another storage area.

**drop-storage-map-statement**
See the DROP STORAGE MAP Statement for a complete description.

**drop-index-statement**
See the DROP INDEX Statement for a complete description.

## Usage Notes

- Before using EXPORT and IMPORT statements, be sure that the database has been backed up in case either the EXPORT or IMPORT statement fails. Use the RMU Backup command. (The IMPORT and EXPORT statements are not the same as the RMU Backup and RMU Restore commands.)
- If you want to restructure an existing database with the EXPORT and IMPORT statements and keep database system files in the same directory, you must use the following sequence:

  1. RMU Backup
  2. EXPORT
  3. DROP DATABASE
  4. IMPORT

  If you do not delete the database, the IMPORT statement fails because the database system files already exist.

OpenVMS VAX OpenVMS Alpha

- When importing the CDD$COMPATIBILITY repository, omit the PATHNAME argument to prevent SQL from attempting to use the repository. ♦
- The CREATE STORAGE AREA, CREATE STORAGE MAP, and CREATE INDEX statements within an IMPORT statement can refer to storage areas, storage maps, and indexes that existed in the original database. When they refer to existing elements, the IMPORT statement implicitly deletes those elements and creates new elements of the same name with the characteristics specified in the CREATE statements (or the database system defaults for characteristics not specified in the CREATE statements).
- The IMPORT statement creates a new database that inherits the characteristics of the database that was the source for the .rbr file used by the IMPORT statement. Only the elements you create will differ from the original database.
- If you do not specify a page size when creating a storage area with the IMPORT statement, the page size is inherited from RDB$SYSTEM.
- If you defined SPAM intervals, you must define them again during an IMPORT operation, otherwise they are set back to the system defaults.

  For a list of defaults for database-wide parameter values and multifile storage area values, see the tables in the *Oracle Rdb7 Guide to Database Performance and Tuning*.
- By default, an IMPORT operation completes three transactions:
  - Creates the new database in a read/write transaction
  - Defines database elements in a read/write transaction
  - Stores the user data using a read/write transaction using exclusive share mode

  If you specify BATCH UPDATE, the IMPORT statement stores the user data and indexes in a batch-update transaction. You cannot recover the database if you use a batch-update transaction.
- To move the database root file, storage areas, and snapshot files to different disks, use the RMU Move_Area command. To move database files to another system, use the RMU Backup and RMU Restore commands. For more information about Oracle RMU commands, see the *Oracle RMU Reference Manual*.

- You can use the IMPORT statement to convert to a multifile database from a single-file database by specifying any CREATE STORAGE AREA clause within the IMPORT statement.
- You can use the IMPORT statement to convert to a single-file database from a multifile database. Use the following steps:
  1. Specify the DROP STORAGE AREA clause for every area in the database, including RDB$SYSTEM. This prevents IMPORT from using the information in the interchange file (.rbr) to define storage areas.

     On OpenVMS, you can use the command RMU Dump Export command with the Nodata qualifier to extract the metadata in the import interchange file to get to see the names of the storage areas in the database. On Digital UNIX, use the rmu -dump -export -nodata command.
  2. Specify the DROP STORAGE MAP clause for every table that contains a storage map.

     Alternately, you could map all tables to the default storage area by specifying the CREATE STORAGE MAP . . . STORE IN RDB$SYSTEM clause.
  3. Specify the DROP INDEX or CREATE INDEX clauses to remove or replace the indexes that are mapped to areas other than RDB$SYSTEM.
  4. Specify the DROP STORAGE MAP clause for the LISTS (segmented string) storage map.
  5. Define the default for LISTS STORAGE AREA to be RDB$SYSTEM.
  6. Define the DEFAULT STORAGE AREA to be RDB$SYSTEM.
- If you specified the PLACEMENT VIA INDEX clause in your storage map definition, a table restored with the IMPORT statement is placed by means of a hashed index. The performance of the IMPORT statement improves when tables are placed by means of a hashed index.
- The RESTRICTED ACCESS clause of the IMPORT statement ensures that other users cannot attach to the database before the IMPORT operation is complete. By default, Oracle Rdb uses the RESTRICTED ACCESS clause on the IMPORT statement.

  Setting restricted access to the database requires DBADM privileges.

An example of the syntax follows:

```
SQL> IMPORT DATABASE FROM temp_test FILENAME new_test RESTRICTED ACCESS;
Exported by Rdb V6.0 Import/Export utility
A component of SQL V6.0
Previous name was test
It was logically exported on 22-SEP-1993 11:43
Multischema mode is DISABLED
Database NUMBER OF USERS is 50
   .
   .
   .
```

If you try to attach to the database before the import operation is complete, you see the following error message:

```
SQL> ATTACH 'FILENAME temp_test';
%SQL-F-ERRATTDEC, Error attaching to database temp_test
-RDB-E-LOCK_CONFLICT, request failed due to locked resource
-RDMS-F-LCKCNFLCT, lock conflict on client
```

- If the NO RESTRICTED ACCESS clause is specified with the IMPORT statement and if users attach to the database before the IMPORT statement has completed, Oracle Rdb displays the following messages:

  ```
  RDO-E-NOIDXREV, unable to import index indexname
  RDB-E-LOCKCONFLICT, request failed ..
  RDB-E-NOMETAUPDATE, ...
  RDMS-F-LCKCONFLCT, lock conflict on client.
  ```

  These messages appear on two of the indexes (both sorted), and the IMPORT statement fails to create those indexes.

  The behavior of the IMPORT statement requires that no other users attach to the database while it is being imported, or alternately, that the indexes are defined in a separate operation.

- See the *Oracle Rdb7 Guide to Database Maintenance* for a complete discussion of when to use the IMPORT, EXPORT, and ALTER DATABASE statements.

- You cannot specify a snapshot file name for a single-file database.

  The SNAPSHOT FILENAME clause specified outside the CREATE STORAGE AREA clause is used to provide a default for subsequent CREATE STORAGE AREA statements. Therefore, this clause does not allow you to create a separate snapshot file for a single-file database (a database without separate storage areas).

When you create a single-file database, Oracle Rdb does not store the file specification of the snapshot file. Instead, it uses the file specification of the root file (.rdb) to determine the file specification of the snapshot file.

If you want to place the snapshot file on a different device or directory, Oracle Rdb recommends that you create a multifile database.

OpenVMS VAX OpenVMS Alpha However, you can work around the restriction on OpenVMS platforms by defining a search list for a concealed logical name. (However, do not use a nonconcealed rooted logical. Database files defined with a nonconcealed rooted logical can be backed up, but do not restore as expected.)

To create a database with a snapshot file on a different device or directory:

1. Define a search list using a concealed logical name. Specify the location of the root file as the first item in the search list and the location of the snapshot file as the second item.
2. Create the database using the logical name for the directory specification.
3. Copy the snapshot file to the second device or directory.
4. Delete the snapshot file from the original location.

If you are doing this with an existing database, close the database using the RMU Close command before defining the search list and open the database using the RMU Open command after deleting the original snapshot file. Otherwise, follow the preceding steps.

An important consideration when placing snapshot and database files on different devices is the process of backing up and restoring the database. Use the RMU Backup command to back up the database. You can then restore the files by executing the RMU Restore command. Copy the snapshot file to the device or directory where you want it to reside and delete the snapshot file from the location to which it was restored. For more information, see the *Oracle RMU Reference Manual*. ♦

- You cannot import a multischema database to earlier versions of Oracle Rdb.

  If an exported database had multischema enabled, the interchange file produced cannot be used by versions of Oracle Rdb earlier than V4.1 to create a similar database. If a limited amount of information is lost, Oracle Rdb recommends exporting the database using the NO EXTENSIONS clause.

Alternatively, the database could be exported and imported within V4.1 by disabling MULTISCHEMA in the SQL IMPORT statement (specifying MULTISCHEMA is OFF). Then, an interchange file produced for the database should be usable in earlier versions.

- The following database definition can cause unexpected input/output to the WORM device and also lead to reduced performance:

  ```
  SQL> IMPORT DATABASE FROM
  cont>   pers FILENAME test_pers
  cont>   LIST STORAGE AREA IS A1
  cont>   CREATE STORAGE AREA A1
  cont>      FILENAME a1
  cont>      WRITE ONCE
  cont>      PAGE FORMAT IS MIXED;
  ```

  This definition requests Oracle Rdb use the WORM storage area as the default list (segmented string) area. That is, all system table lists will be stored in the WORM storage area. Because the type of segmented strings written for the system metadata are often revised (for example, a COMMENT IS, or an ALTER statement) and often accessed at run time, they are unsuited for storage on a WORM device.

  Oracle Rdb issues the following message when you attempt to use a WORM storage area as the default list storage area:

```
%SQL-F-ERRCRESCH, Error creating database filename test_pers
-RDB-E-BAD_DPB_CONTENT, invalid database parameters in the database parameter block (DPB)
-RDMS-E-DEFLISTWORM, default list (segmented string) storage area can not be a WRITE ONCE area
```

- Oracle Rdb is not supported on Distributed File Server (DFS) disks. Because DFS does not support shared write, you cannot import a database to a DFS-mounted disk. Oracle Rdb requires shared access because both the monitor and user need to open the root file simultaneously.
- The IMPORT statement is compatible with succeeding versions of Oracle Rdb. For example, you can import a database using a higher version of Oracle Rdb than the version used to create the database you are importing. You cannot import a database using a lower version of Oracle Rdb.
- In you have created a database specifying the SYSTEM INDEX COMPRESSION clause, you can change the compression mode during an import operation. For example, if you created a database specifying the SYSTEM INDEX COMPRESSION IS DISABLED, you can specify SYSTEM INDEX COMPRESSION IS ENABLED during an import operation.

- Oracle Rdb does not recalculate the asynchronous prefetch DEPTH BUFFERS, the asynchronous batch write CLEAN BUFFER COUNT, or the asynchronous batch write MAXIMUM BUFFER COUNT when you import a database, even if you specify a value for the NUMBER OF BUFFER clause. Oracle Rdb uses the values from the export operation, unless you specify values for each clause.
- If you specify values for asynchronous prefetch DEPTH BUFFERS, the asynchronous batch write CLEAN BUFFER COUNT, or the asynchronous batch write MAXIMUM BUFFER COUNT, Oracle Rdb checks that the values are within appropriate limits.
- OpenVMS VAX OpenVMS Alpha If a database contains a table with a computed-by column that is dependent on another computed-by column in another table, the import operation completes but all computed-by columns in the group fail to be created. Oracle Rdb cannot define that computed-by column because the computed-by column that is referenced is not yet defined.

  The following are workarounds to this problem:

  – Redefine the computed-by columns so that there are no references to other computed-by columns.

  – Before exporting the database, delete all computed-by columns that reference other computed-by columns. After the import operation, re-create these computed-by columns. To simplify this procedure, you can create a short SQL script to run before each export operation and after each import operation that accomplishes this action. ♦
- Because of some special characteristics of the Norwegian collating sequence, certain restrictions apply when creating a Norwegian collating sequence in a database. The name of a Norwegian collating sequence in the NCS library must begin with the character string NORWEGIAN.

  The sequence customarily shipped with OpenVMS is named NORWEGIAN, which meets this restriction. You may wish to alter the Norwegian sequence slightly or change its name. Oracle recommends that any variation of the Norwegian collating sequence be given a name such as NORWEGIAN_1 or NORWEGIANA.
- CREATE CACHE does not assign the row cache area to a storage area. You must use the CACHE USING clause with the CREATE STORAGE AREA clause of the CREATE DATABASE statement or the CACHE USING clause with the ADD STORAGE AREA or ALTER STORAGE AREA clauses of the ALTER DATABASE statement.

- The product of the CACHE SIZE and the ROW LENGTH settings determine the amount of memory required for the row cache area (some additional overhead and rounding up to page boundaries is performed by the database system).
- The row cache area is shared by all processes attached to the database on any node.
- Oracle Rdb recommends that you specify the UNIFORM page format for improved performance when specifying a default storage area.
- You cannot delete a storage area that has been established as the database default storage area.
- You cannot enable after-image journaling or add after-image journal files with the IMPORT statement. You must use the ALTER DATABASE statement to enable after-image journaling or add after-image journal files.
- After-image journal attributes cannot be imported and are disabled after IMPORT completes. Therefore, fast commit is also disabled.

  Prior to executing the EXPORT statement, use the RMU Extract Item=Alter_Database command to generate a script of the after-image journal definition. Once the database has been exported and imported, run the script against the imported database to re-create the original after-image journal attributes. See the *Oracle RMU Reference Manual* for more information on the RMU Extract command.

## Examples

Example 1: Converting to a multifile database

This example uses the EXPORT and IMPORT statements to convert the online sample database, personnel, to a multifile database.

```
$ SQL
SQL> EXPORT DATABASE FILENAME personnel INTO pers;
SQL> IMPORT DATABASE FROM
cont> pers FILENAME mf_personnel
cont> CREATE STORAGE AREA MFP1 FILENAME mfp1
cont> CREATE STORAGE AREA MFP2 FILENAME mfp2
cont> CREATE STORAGE MAP EMPLOYEE_MAP FOR EMPLOYEES
cont>   STORE RANDOMLY ACROSS (mfp1, mfp2);
Exported by Rdb V6.0 Import/Export utility
A component of SQL V6.0
Previous name was bench:personnel
It was logically exported on  8-SEP-1993 16:36
```

```
Database NUMBER OF USERS is 50
Database NUMBER OF CLUSTER NODES is 16
Database NUMBER OF DBR BUFFERS is 20
Database SNAPSHOT is ENABLED
Database SNAPSHOT is IMMEDIATE
Database BUFFER SIZE is 6 blocks
Database NUMBER OF BUFFERS is 20
Adjustable lock granularity is ENABLED
Database global buffering is DISABLED
Journal checkpointing is DISABLED
Journal FAST COMMIT is DISABLED
LOCK TIMEOUT is 0 seconds
IMPORTing STORAGE AREA: RDB$SYSTEM
IMPORTing table COLLEGES
IMPORTing table DEGREES
IMPORTing table DEPARTMENTS
IMPORTing table EMPLOYEES
IMPORTing table JOBS
IMPORTing table JOB_HISTORY
IMPORTing table SALARY_HISTORY
IMPORTing table WORK_STATUS
IMPORTing view CURRENT_SALARY
IMPORTing view CURRENT_JOB
IMPORTing view CURRENT_INFO
```

Example 2: Importing a database created with ANSI/ISO-style privileges

This example imports a database originally created with ANSI/ISO-style privileges as an ACL-style database.

```
SQL> IMPORT DATABASE WITH ALIAS EXAMPLE
cont> FROM ansi_example
Exported by Rdb V6.0 Import/Export utility
A component of SQL V6.0
Previous name was foo
It was logically exported on 15-SEP-1993 17:17
Database NUMBER OF USERS is 50
Database NUMBER OF CLUSTER NODES is 16
Database NUMBER OF DBR BUFFERS is 20
Database SNAPSHOT is ENABLED
Database SNAPSHOT is IMMEDIATE
Database BUFFER SIZE is 6 blocks
Database NUMBER OF BUFFERS is 20
Adjustable lock granularity is ENABLED
Database global buffering is DISABLED
Journal checkpointing is DISABLED
Journal FAST COMMIT is DISABLED
LOCK TIMEOUT is 0 seconds
IMPORTing STORAGE AREA: RDB$SYSTEM
```

```
SQL> --
SQL> -- The newly imported ACL-style database now has
SQL> -- the following ACL-style privileges:
SQL> --
SQL> SHOW PROTECTION ON DATABASE ALIAS example;
Protection on Alias EXAMPLE
    (IDENTIFIER=[sql,warring],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+
      DROP+OPERATOR+DBADM+REFERENCES)
```

Example 3: Importing a database and displaying statistics

This example imports a database and uses the TRACE option to display DIO, CPU, and PAGE FAULT statistics.

```
SQL> IMPORT DATABASE FROM personnel.rbr
cont>  FILENAME personnel_new.rdb
cont>  TRACE
cont>  CREATE INDEX LOCAL_INDEX ON jobs (job_code);
Exported by Rdb V6.0 Import/Export utility
A component of SQL V6.0
Previous name was personnel
It was logically exported on  2-SEP-1993 16:23
Database NUMBER OF USERS is 50
Database NUMBER OF CLUSTER NODES is 16
Database NUMBER OF DBR BUFFERS is 20
Database SNAPSHOT is ENABLED
Database SNAPSHOT is IMMEDIATE
Database BUFFER SIZE is 6 blocks
Database NUMBER OF BUFFERS is 20
Adjustable lock granularity is ENABLED
Database global buffering is DISABLED
Journal fast commit is DISABLED
Journal fast commit checkpoint interval is 0 blocks
Journal fast commit checkpoint time is 0 seconds
Commit to journal optimization is Disabled
Journal fast commit TRANSACTION INTERVAL is 256
```

```
LOCK TIMEOUT is 0 seconds
IMPORTing STORAGE AREA: RDB$SYSTEM
IMPORTing table COLLEGES
Completed COLLEGES. DIO = 103, CPU = 0:00:00.89, FAULTS = 169
Starting INDEX definition COLL_COLLEGE_CODE
Completed COLL_COLLEGE_CODE. DIO = 25, CPU = 0:00:00.24, FAULTS = 26
IMPORTing table DEGREES
Completed DEGREES. DIO = 96, CPU = 0:00:01.15, FAULTS = 9
Starting INDEX definition DEG_COLLEGE_CODE
Completed DEG_COLLEGE_CODE. DIO = 27, CPU = 0:00:00.36, FAULTS = 1
Starting INDEX definition DEG_EMP_ID
Completed DEG_EMP_ID. DIO = 39, CPU = 0:00:00.49, FAULTS = 2
IMPORTing table DEPARTMENTS
Completed DEPARTMENTS. DIO = 99, CPU = 0:00:00.70, FAULTS = 3
IMPORTing table EMPLOYEES
Completed EMPLOYEES. DIO = 182, CPU = 0:00:01.60, FAULTS = 21
   .
   .
   .
Starting CONSTRAINT definition SH_EMPLOYEE_ID_IN_EMP
Completed SH_EMPLOYEE_ID_IN_EMP. DIO = 48, CPU = 0:00:00.56, FAULTS = 2
Starting CONSTRAINT definition WS_STATUS_CODE_DOM_NOT_NULL
Completed WS_STATUS_CODE_DOM_NOT_NULL. DIO = 36, CPU = 0:00:00.23, FAULTS = 0
Completed import. DIO = 3530, CPU = 0:00:32.97, FAULTS = 2031
SQL>
```

## INCLUDE Statement

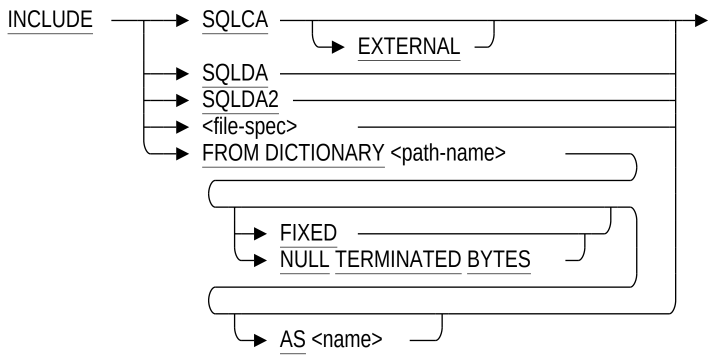
Inserts declarations or code into a precompiled host language program. You can use the INCLUDE statement to insert:

- Host language declarations for the SQL Communications Area (SQLCA) and a message vector
- Host language declarations for the SQL Descriptor Areas (SQLDA and SQLDA2)
- Host language source code
- OpenVMS VAX OpenVMS Alpha Host language declarations for repository record definitions ♦

### Environment

You can issue the INCLUDE statement embedded in precompiled host language programs only. Programs must either use an INCLUDE SQLCA statement or explicitly declare an SQLCODE variable. The other forms of the INCLUDE statement are optional (see the Usage Notes).

### Format



### Arguments

**SQLCA**
Specifies that SQL inserts into the program the SQLCA and a message vector (RDB$MESSAGE_VECTOR) structure specific to supported database systems. Both the SQLCA and the message vector provide ways of handling error conditions:

- The SQLCA is a collection of variables that SQL uses to provide information about the execution of SQL statements to application programs. The SQLCA shows if a statement was successful and, for some conditions, the particular error when a statement was not successful.
- The message vector is also a collection of variables that SQL updates after SQL executes a statement. The message vector also lets programs check if a statement was successful, but provides more detail than the SQLCA about the type of error condition if a statement was not successful.

For more information on the SQLCA and the message vector, see Appendix B.

**EXTERNAL**
Declares an external reference to the SQLCA structure for SQL precompiled C programs. If you have multiple modules that use the INCLUDE SQLCA statement, you can add the EXTERNAL keyword to all but one of them.

On OpenVMS, if your application shares the SQLCA among multiple images, one image must define the SQLCA while all other images must reference the SQLCA. Use the EXTERNAL keyword to reference the SQLCA.

On Digital UNIX, you must have at most one definition of the SQLCA, regardless of whether or not your application uses multiple images.

**SQLDA**
Specifies that SQL inserts the SQLDA into the program. The SQLDA is a collection of variables used only in dynamic SQL. The SQLDA provides information about dynamic SQL statements to the program, and information about host language variables in the program to SQL.

**SQLDA2**
Specifies that SQL inserts the SQLDA2 into the program. The SQLDA2, like the SQLDA, is a collection of variables that provides information about dynamic SQL statements to the program and information about host language variables in the program to SQL. You should use the SQLDA2 in any dynamic statement where the column name used in a parameter marker or select list item is one of the date-time or interval data types.

For more information on the SQLDA and SQLDA2, see Appendix D.

**file-spec**
The file specification for source code to be inserted into your program. The file specification must refer to a standard OpenVMS text file. SQL does not support the INCLUDE statement from text libraries (file extension .tlb). Use the SQL INCLUDE statement in either of these cases:

- The source code to be included contains embedded SQL statements.

- The source code to be included contains host language variable declarations to which embedded SQL statements in other parts of the program refer.

If the source code contains neither SQL statements nor variables to which SQL statements refer, using the SQL INCLUDE statement is no different from using host language statements to include files.

OpenVMS VAX OpenVMS Alpha

**FROM DICTIONARY path-name**
Specifies the path name for a repository record definition. Because SQL treats the path name as a string literal, you should enclose it in single quotation marks. SQL declares a host structure corresponding to the repository record definition and gives it the same name. SQL statements embedded in the program can then refer to the host structure.

Typically, programs use the FROM DICTIONARY argument as a convenient way to declare host structures that correspond to table definitions stored in the repository.

SQL stores table definitions in the repository in the following cases only:

- Both the CREATE DATABASE statement and the database declaration for the attach in which the table was defined specified the PATHNAME argument.
- The database definitions were copied to the repository with an INTEGRATE statement.

However, programs can use the FROM DICTIONARY argument to declare host structures for any CDD$RECORD repository object type, whether those repository objects were defined as part of the database.

Using the INCLUDE statement does more than using a comparable host language statement that inserts a CDD$RECORD object into the program. The INCLUDE FROM DICTIONARY statement lets you refer to the repository record in an embedded SQL statement, while the host language statement does not.

The FROM DICTIONARY clause is available only on OpenVMS platforms. ♦

OpenVMS VAX OpenVMS Alpha

**FIXED**
The FIXED and NULL TERMINATED BYTES clauses tell the precompiler how to interpret C language CHAR fields. If you specify FIXED, the precompiler interprets CHAR fields from the repository as fixed character strings.

The FIXED clause is available only on OpenVMS platforms. ♦

OpenVMS VAX OpenVMS Alpha

**NULL TERMINATED BYTES**
Specifies that CHAR fields from the repository are null-terminated. The module processor interprets the length field in the repository as the number of bytes in the string. If *n* is the length in the repository, then the number of data bytes is *n–1*, and the length of the string is *n* bytes.

In other words, the precompiler assumes that the last character of the string is for the null terminator. Thus, a field that the repository lists as 10 characters can only hold a 9-character SQL field from the C precompiler.

If you do not specify a character interpretation option, NULL TERMINATED BYTES is the default.

For more information, see the NULL TERMINATED CHARACTERS argument in Chapter 3.

The NULL TERMINATED BYTES clause is available only on OpenVMS platforms. ♦

OpenVMS VAX OpenVMS Alpha

**AS name**
Specifies a name to override the structure name of the record from the repository. By default, the SQL precompiler takes the structure name from the repository record name.

The AS clause can be used only in embedded C programs and is available only on OpenVMS platforms. ♦

## Usage Notes

OpenVMS VAX OpenVMS Alpha

- The Ada and Pascal precompilers do not support the INCLUDE FROM DICTIONARY statement. ♦
- You do not have to use the INCLUDE SQLCA statement in programs. However, if you do not, you must explicitly declare the SQLCODE variable to receive values from SQL.

  To comply with the ANSI/ISO SQL standard, you should explicitly declare the SQLCODE variable instead of using the INCLUDE SQLCA statement. However, programs that do not use the INCLUDE SQLCA statement will not have the RDB$MESSAGE_VECTOR message vector structure declared by the precompiler. Such programs may have to explicitly declare the message vector. See Section B.3 for sample declarations of the message vector.
- Programs that use an INCLUDE SQLCA statement must place it where it is valid to declare variables.

- All SQL statements embedded in a precompiled program must be within the scope of either an SQLCODE or SQLCA declaration. However, the SQL precompiler supports block structure in Pascal, Ada, and C programs but not in COBOL, FORTRAN, or PL/I. This means SQL is more restrictive about where it allows embedded SQL statements in COBOL, FORTRAN, and PL/I programs that contain multiple modules than in Pascal, Ada, and C (a module is a set of statements that can be separately compiled).
  - In COBOL, FORTRAN, and PL/I programs, only one module can declare an SQLCA or SQLCODE parameter. Because of this, program source files with more than one module cannot contain embedded SQL statements in more than one of the modules.

    If a module contains more than one routine, you can use SQL statements in those routines provided they are within the scope of the INCLUDE SQLCA statement. COBOL and PL/I allow such nested routines, but FORTRAN does not.
  - In Ada, C, and Pascal programs, all SQL statements must be within the scope of an SQLCODE or SQLCA declaration; however, each module of a program can contain a declaration (or many declarations, such as one in each routine in the module). Thus, you can embed SQL statements in more than one module in Ada, C, and Pascal programs.
- OpenVMS VAX OpenVMS Alpha SQL does not require programs that use the INCLUDE FROM DICTIONARY statement to declare aliases with the PATHNAME argument. However, programs that use the INCLUDE FROM DICTIONARY statement to declare host structures that correspond to table definitions must specify a complete repository path name for those table definitions.

  The database system stores table definitions in a path name called RDB$RELATIONS that is subordinate to the database path name. The path name in the INCLUDE FROM DICTIONARY statement must include the RDB$RELATIONS name in the path name specification. ♦
- Source code files specified in an SQL INCLUDE file-spec statement cannot contain nested INCLUDE file-spec statements themselves.
- The SQL precompiler will not process an INCLUDE statement in the middle of a variable declaration. The following segment from a COBOL program illustrates an INCLUDE statement that is not processed:

```
01     dept_rec     pic x(24)

01     commarea.

EXEC SQL  INCLUDE 'A.DAT' END-EXEC
```

Digital UNIX

- Only the C, COBOL, FORTRAN, and Pascal languages are supported on Digital UNIX. ♦

## Examples

OpenVMS VAX OpenVMS Alpha

Example 1: Including a host structure declaration

This simple COBOL program uses the INCLUDE FROM DICTIONARY statement to declare a host structure that corresponds to the EMPLOYEES table in the sample personnel database. The repository path name must specify the RDB$RELATIONS repository directory between the database directory and the table name.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INCLUDE_FROM_CDD.
*
* Illustrate how to use the INCLUDE FROM DICTIONARY
* statement to declare a host structure corresponding to
* the EMPLOYEES table:
*
DATA DIVISION.
WORKING-STORAGE SECTION.
EXEC SQL WHENEVER SQLERROR GOTO ERR END-EXEC.
*
* Include the SQLCA:
EXEC SQL INCLUDE SQLCA END-EXEC.
*
* Declare the schema:
* (Notice that declaring the alias with the
* FILENAME qualifier would not have precluded
* using the INCLUDE FROM DICTIONARY statement later.)
EXEC SQL DECLARE PERS ALIAS FOR
        PATHNAME 'CDD$DEFAULT.PERSONNEL' END-EXEC.
```

```
*
* Create a host structure that corresponds to the
* EMPLOYEES table with the INCLUDE FROM DICTIONARY
* statement.  The path name in the INCLUDE statement
* must specify the RDB$RELATIONS directory before
* the table name:
EXEC SQL INCLUDE FROM DICTIONARY
        'CDD$DEFAULT.PERSONNEL.RDB$RELATIONS.EMPLOYEES'
        END-EXEC.
*
* Declare an indicator structure for the host
* structure created by the INCLUDE FROM DICTIONARY statement:
01 EMPLOYEES-IND.
        02 EMP-IND OCCURS 12 TIMES PIC S9(4) COMP.
EXEC SQL DECLARE E_CURSOR CURSOR
        FOR SELECT * FROM EMPLOYEES  END-EXEC.

PROCEDURE DIVISION.
0.
    DISPLAY "Display rows from EMPLOYEES:".
    EXEC SQL OPEN E_CURSOR END-EXEC
    EXEC SQL FETCH E_CURSOR INTO :EMPLOYEES:EMP-IND  END-EXEC
    PERFORM UNTIL SQLCODE NOT = 0
        DISPLAY EMPLOYEE_ID, FIRST_NAME, LAST_NAME
        EXEC SQL FETCH E_CURSOR INTO :EMPLOYEES:EMP-IND END-EXEC
    END-PERFORM
    EXEC SQL CLOSE E_CURSOR END-EXEC

    EXEC SQL ROLLBACK END-EXEC.
    EXIT PROGRAM.

ERR.
    DISPLAY "unexpected error ", sqlcode with conversion.
    CALL "SQL$SIGNAL".
```

♦

OpenVMS VAX OpenVMS Alpha

Example 2: Including the SQLCA

This fragment from a PL/I program shows the INCLUDE SQLCA statement and illustrates how an error-handling routine refers to the SQLCA.

The program creates an intermediate result table, TMP, and copies the EMPLOYEES table from the personnel database into it. It then declares a cursor for TMP and displays the rows of the cursor on the terminal screen.

```
    /* Include the SQLCA: */
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL WHENEVER SQLERROR GOTO ERROR_HANDLER;
    EXEC SQL DECLARE ALIAS FOR FILENAME personnel;
    DCL MANAGER_ID CHAR(5),
        LAST_NAME CHAR(20),
        DEPT_NAME CHAR(20);
    DCL COMMAND_STRING CHAR(256);

    EXEC SQL CREATE TABLE TMP
            (MANAGER_ID CHAR(5),
             LAST_NAME CHAR(20),
             DEPT_NAME CHAR(20));
    COMMAND_STRING =
        'INSERT INTO TMP
                SELECT  E.LAST_NAME,
                        E.FIRST_NAME,
                        D.DEPARTMENT_NAME
                FROM EMPLOYEES E, DEPARTMENTS D
                WHERE E.EMPLOYEE_ID = D.MANAGER_ID';

    EXEC SQL EXECUTE IMMEDIATE :COMMAND_STRING;

    EXEC SQL DECLARE X CURSOR FOR SELECT * FROM TMP;
    EXEC SQL OPEN X;
    EXEC SQL FETCH X INTO MANAGER_ID, LAST_NAME, DEPT_NAME;
    DO WHILE (SQLCODE = 0);
        PUT SKIP EDIT
                (MANAGER_ID, ' ', LAST_NAME, ' ', DEPT_NAME)
                (A,A,A,A,A);
        EXEC SQL FETCH X INTO MANAGER_ID, LAST_NAME, DEPT_NAME;
    END;
    EXEC SQL ROLLBACK;
    PUT SKIP EDIT ('  ALL OK') (A);
    RETURN;

ERROR_HANDLER:

    /* Display the value of the SQLCODE field in the SQLCA: */
    PUT SKIP EDIT ('UNEXPECTED SQLCODE VALUE ', SQLCODE) (A, F(9));
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
```
♦

## INSERT Statement

Adds a new row, or a number of rows, to a table or view. You can also use the INSERT statement with a cursor to assign values to the segments in a column of the LIST OF BYTE VARYING data type.

Before you assign values to the segments in a column of the LIST OF BYTE VARYING data type, you must first assign a value to one or more other columns in the same row. To do this, use a positioned insert. A **positioned insert** is an INSERT statement that specifies an insert-only table cursor. This type of INSERT statement sets up the proper row context for subsequent list cursors to assign values to list segments.

You can specify the name of a static, a dynamic, or an extended dynamic cursor in a positioned insert. If you specify a static cursor name, that cursor name must also be specified in a DECLARE CURSOR statement within the same module. See the DECLARE CURSOR Statement for more information on static, dynamic, and extended dynamic cursors.

When you use an INSERT statement to assign values to list segments:

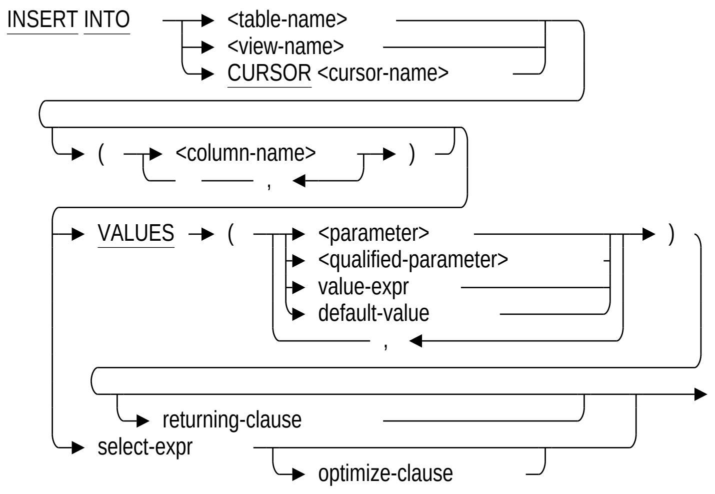
- The current transaction must not be read-only.
- You cannot specify a cursor name that refers to an update table cursor.
- Your cursor must specify an intermediate table.
- The value that you assign is appended to the end of the list.

### Environment

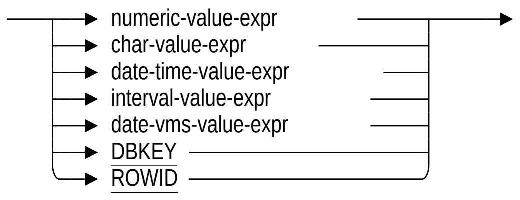
You can use the INSERT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

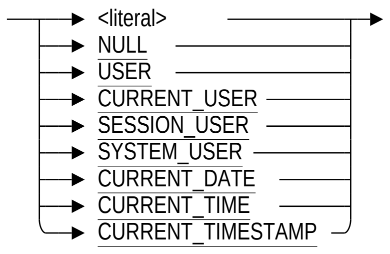
## Format



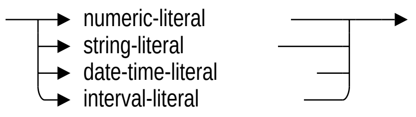
value-expr =



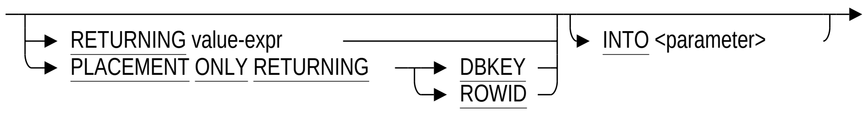
default-value =



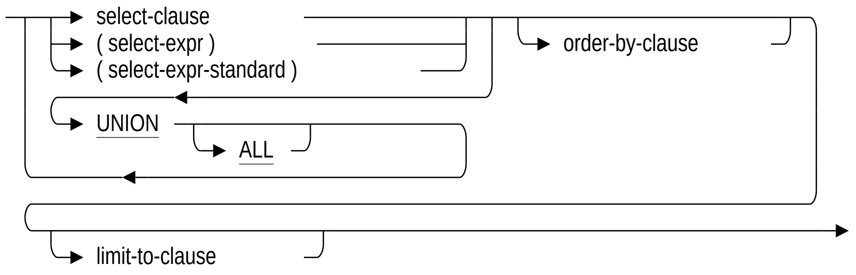
literal =



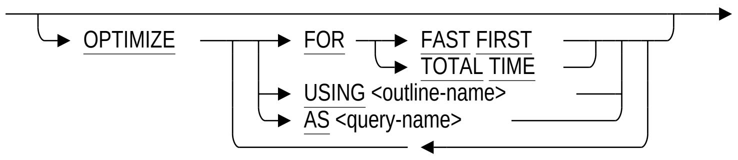
returning-clause =



select-expr =



optimize-clause =



## Arguments

**INTO table-name**
**INTO view-name**
The name of the target table or view to which you want to add a row.

**CURSOR cursor-name**
Keyword required when using cursors. You must use a cursor to insert values into any row that contains a column of the LIST OF BYTE VARYING data type.

### column-name

Specifies a list of names of columns in the table or view. You can list the columns in any order, but the names must correspond to those of the table or view.

If you do not include all the column names in the list, SQL assigns a null value to those not specified, unless columns were:

- Defined with a default
- Based on a domain that has a default
- Defined with the NOT NULL clause in the CREATE TABLE statement

You cannot omit from an INSERT statement the names of columns defined with the NOT NULL clause. If you do, the statement fails.

Omitting the list of column names altogether is the same as listing all the columns of the table or view in the same order as they were defined.

You must omit the list of column names when using the INSERT statement to assign values to the segments in a column of data type LIST OF BYTE VARYING. Column names are not valid in this context.

### VALUES value-expr

Specifies a list of values to be added to the table as a single row. The values can be specified through parameters, qualified parameters, column select expressions, value expressions, or the default values: NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP.

See Chapter 2 for more information about parameters, qualified parameters, column select expressions, value expressions, and default values.

The values listed in the VALUES argument can be selected from another table, but both tables must reside in the same database.

The number of values in the list must correspond to the number of columns specified in the list of column names. If you did not specify a column list, the number of values in the list must be the same as the number of columns in the table. The first value specified in the list is assigned to the first column, the second value to the second column, and so on.

You cannot specify the NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, or CURRENT_ TIMESTAMP keywords in an INSERT statement used to assign values to the segments in a column of the LIST OF BYTE VARYING data type. You cannot assign NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP values to a column of the LIST OF BYTE VARYING data type.

See the SQL Online Help topic INSERT EXAMPLES for an example that shows an INSERT statement with a column select expression.

**RETURNING value-expr**
Returns the value of the column specified in the values list. If DBKEY or ROWID is specified, this argument returns the database key (dbkey) of the row being added. (The ROWID keyword is a synonym to the DBKEY keyword.) When the DBKEY value is valid, subsequent queries can use the DBKEY value to access the row directly.

The RETURNING DBKEY clause is not valid in an INSERT statement used to assign values to the segments in a column of the LIST OF BYTE VARYING data type.

**PLACEMENT ONLY RETURNING DBKEY**
**PLACEMENT ONLY RETURNING ROWID**
Returns the dbkey of a specified record, but does not insert any actual data. The PLACEMENT ONLY RETURNING DBKEY clause (or the PLACEMENT ONLY RETURNING ROWID clause) lets you determine the target page number for records that are to be loaded into the database. (The keyword ROWID is a synonym to the DBKEY keyword.) When you use this clause, only the area and page numbers from the dbkeys are returned. Use of this clause can improve bulk data loads. If you use the PLACEMENT ONLY clause, you can return only the dbkey values. Use the PLACEMENT ONLY RETURNING DBKEY clause only in programs that load data into an existing database and only with rows placed via a hashed index in the storage map. For more information, see the *Oracle Rdb7 Guide to Database Design and Definition*.

**INTO parameter**
Inserts the value specified to a specified parameter. The INTO parameter clause is not valid in interactive SQL.

**select-expr**
Specifies a select expression that specifies a result table. The result table can contain zero or more rows. All the rows of the result table are added to the target table named in the INTO clause.

This is the only situation supported in SQL that allows you to specify a second database in a single SQL statement.

The number of columns in the result table must correspond to the number of columns specified in the list of column names. If you did not specify a list of column names, the number of columns in the result table must be the same as the number of columns in the target table. For each row of the result table, the value of the first column is assigned to the first column of the target table, the second value to the second column, and so on.

You cannot specify a select expression in an INSERT statement used to assign values to the segments in a column of the LIST OF BYTE VARYING data type.

For detailed information on select expressions, see Section 2.8.1.

**OPTIMIZE FOR**
The OPTIMIZE FOR clause specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- FAST FIRST

  A query optimized for FAST FIRST returns data to the user as quickly as possible, even at the expense of total throughput.

  If a query can be cancelled prematurely, you should specify FAST FIRST optimization. A good candidate for FAST FIRST optimization is an interactive application that displays groups of records to the user, where the user has the option of aborting the query after the first few screens. For example, singleton SELECT statements default to FAST FIRST optimization.

  If optimization strategy is not explicitly set, FAST FIRST is the default.

- TOTAL TIME

  If your application runs in batch, accesses all the records in the query, and performs updates or writes a report, you should specify TOTAL TIME optimization. Most queries benefit from TOTAL TIME optimization.

**OPTIMIZE USING outline-name**
The OPTIMIZE USING clause explicitly names the query outline to be used with the select expression even if the outline ID for the select expression and for the outline are different.

See the CREATE OUTLINE Statement for more information on creating an outline.

**OPTIMIZE AS query-name**

The OPTIMIZE AS clause assigns a name to the query. You must define the RDMS$DEBUG_FLAGS logical name or the RDB_DEBUG_FLAGS configuration parameter be the letter *S* to see the access methods used to produce the results of the query.

## Usage Notes

- Avoid issuing a statement that creates an infinite number of inserts. For example:

  ```
  SQL> INSERT INTO DEPARTMENTS
  cont>  (DEPARTMENT_NAME, BUDGET_ACTUAL)
  cont>  VALUES
  cont>  ('RECR',
  cont>   128776);
  1 row inserted
  SQL> INSERT INTO DEPARTMENTS
  cont>     (DEPARTMENT_NAME,
  cont>      BUDGET_ACTUAL)
  cont>      SELECT DEPARTMENT_NAME, BUDGET_ACTUAL + 1 FROM DEPARTMENTS;
  %RDB-E-ARITH_EXCEPT, Truncation of numeric value at runtime
  -SYSTEM-F-INTOVF , ARITHMETIC TRAP, INTEGER OVERFLOW AT PC=0028E3A3,
  PSL=0140000
  ```

  SQL scans the columns in the table sequentially, and because the preceding INSERT statement adds a new column at the end of the table, the query never reaches the end. You can avoid this error in either of two ways:

  – Specify a selection criteria on the SELECT part of the INSERT statement. One method is to create an index on one of the columns and then add a WHERE BETWEEN clause.

  – Use the SET QUERY LIMIT statement to restrict the output generated. For more information, see the SET Statement.

- When you use the INSERT statement to add rows to a view, you are actually adding rows to the base tables on which the view is based. Because you can get unexpected results by inserting rows into views, be careful using the INSERT statement with views. In addition, SQL restricts the types of views with which you can use the INSERT statement. See the CREATE VIEW Statement for rules about inserting, updating, and deleting values in views.

- You can get a confusing error message when you attempt to insert rows into a view and both the following are true:
  - The view is based on a table that contains a column defined with the NOT NULL attribute.
  - The view definition does not include the column defined with the NOT NULL attribute.

  For example:

```
SQL> -- Create a view that is not a read-only view:
SQL> CREATE VIEW TEMP AS
cont>   SELECT SUPERVISOR_ID FROM JOB_HISTORY;
SQL>
SQL> -- However, the JOB_HISTORY table on which the view is based
SQL> -- contains a column, EMPLOYEE_ID, that is defined with the
SQL> -- NOT NULL attribute.  Because the TEMP view does not include
SQL> -- the EMPLOYEE_ID column, all attempts to insert rows into
SQL> -- it will fail:
SQL> INSERT INTO TEMP (SUPERVISOR_ID) VALUES ('99999');
1 row inserted
SQL> COMMIT;
%RDB-E-INTEG_FAIL, violation of constraint JH_EMPLOYEE_ID_IN_EMP
     caused operation to fail
SQL> ROLLBACK;
```

- To move data between databases, SQL lets you refer to a table from one database in the INTO clause of an INSERT statement, and to tables from another database in a select expression within that INSERT statement.

  This is the only situation supported in SQL that allows you to specify a second database in a single SQL statement. Example 4 illustrates this point.

- The PLACEMENT ONLY RETURNING DBKEY (or ROWID) clause of the INSERT statement returns the dbkey of a specified row. This clause allows an application to build a list of unordered dbkeys for all specified rows. You can then use the Sort utility (SORT) to create a sorted list of dbkeys and use this sorted list to insert the rows. When you store records sorted by dbkey, you are writing rows to database pages in sequence with all rows for a page written to the page while it is in the buffer. Because less random I/O is involved when you store records in this way, a significant performance improvement can occur during your load procedure. This clause can result in significant performance improvements in database load procedures that specify the PLACEMENT VIA INDEX clause for a hashed index. Use it only with records for which a hashed index has been defined.

- You cannot insert a row into an insert-only table cursor by using the RETURNING DBKEY clause.

  The following example shows the invalid syntax:

  ```
  SQL> ATTACH 'FILENAME MF_PERSONNEL';
  SQL> DECLARE CURSOR1 INSERT ONLY TABLE CURSOR FOR SELECT * FROM COLLEGES;
  SQL> OPEN CURSOR1;
  SQL> INSERT INTO CURSOR CURSOR1 (COLLEGE_CODE, COLLEGE_NAME)
  cont> VALUES ('ASU','Arizona State University') RETURNING DBKEY;
  %SQL-F-NORETURN, Specifying a RETURNING clause is incompatible with a
  positioned insert statement
  SQL> CLOSE CURSOR1;
  SQL>
  SQL> DECLARE CURSOR2 INSERT ONLY TABLE CURSOR FOR
  cont> SELECT * FROM RESUMES;
  SQL> OPEN CURSOR2;
  SQL> INSERT INTO CURSOR CURSOR2 (EMPLOYEE_ID)
  cont> VALUES ('00169') RETURNING DBKEY;
  %SQL-F-NORETURN, Specifying a RETURNING clause is incompatible with a
  positioned insert statement
  SQL> CLOSE CURSOR2;
  SQL> DISCONNECT ALL;
  ```

  To avoid this problem, specify the SQL INSERT statement without using a cursor. Use the INSERT INTO table-name . . . RETURNING DBKEY INTO . . . syntax.

- If an outline exists, Oracle Rdb uses the outline specified in the OPTIMIZE USING clause unless one or more of the directives in the outline cannot be followed. For example, if the compliance level for the outline is mandatory and one of the indexes specified in the outline directives has been deleted, the outline is not used. SQL issues an error message if an existing outline cannot be used.

  If you specify the name of an outline that does not exist, Oracle Rdb compiles the query, ignores the outline name, and searches for an existing outline with the same outline ID as the query. If an outline with the same outline ID is found, Oracle Rdb attempts to execute the query using the directives in that outline. If an outline with the same outline ID is not found, the optimizer selects a strategy for the query for execution.

  See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information regarding query outlines.

## Examples

Example 1: Adding a row with literal values

The following interactive SQL example stores a new row in the DEPARTMENTS table of the sample personnel database. It explicitly assigns a literal value to each column in the row. Because the statement includes the RETURNING DBKEY clause, SQL returns the dbkey value 29:435:9.

```
SQL> INSERT INTO DEPARTMENTS
cont>   -- List of columns:
cont>    (DEPARTMENT_CODE,
cont>     DEPARTMENT_NAME,
cont>     MANAGER_ID,
cont>     BUDGET_PROJECTED,
cont>     BUDGET_ACTUAL)
cont> VALUES
cont>   -- List of values:
cont>    ('RECR',
cont>     'Recreation',
cont>     '00175',
cont>     240000,
cont>     128776)
cont> RETURNING DBKEY;
                  DBKEY
              29:435:9
1 row inserted
```

Example 2: Adding a row using parameters

This example is a COBOL program fragment that adds a row to the JOB_ HISTORY table by explicitly assigning values from parameters to columns in the table. This example:

- Prompts for the column values.
- Declares a read/write transaction. Because you are updating the JOB_ HISTORY table, you do not want to conflict with other users who may be reading data from this table. Therefore, you use the protected share mode and the write lock type.
- Stores the row by assigning the parameters to the columns of the table.
- Checks the value of the SQLCODE variable and repeats the INSERT operation if the value is less than zero.

- Uses the COMMIT statement to make the update permanent.

```
STORE-JOB-HISTORY.

   DISPLAY "Enter employee ID:       " WITH NO ADVANCING.
   ACCEPT EMPL-ID.
   DISPLAY "Enter job code:          " WITH NO ADVANCING.
   ACCEPT JOB-CODE.
   DISPLAY "Enter starting date:     " WITH NO ADVANCING.
   ACCEPT START-DATE.
   DISPLAY "Enter ending date:       " WITH NO ADVANCING.
   ACCEPT END-DATE.
   DISPLAY "Enter department code:   " WITH NO ADVANCING.
   ACCEPT DEPT-CODE.
   DISPLAY "Enter supervisor’s ID:   " WITH NO ADVANCING.
   ACCEPT SUPER.

EXEC SQL
        SET TRANSACTION READ WRITE
                RESERVING JOB_HISTORY FOR PROTECTED WRITE
END-EXEC

EXEC SQL
        INSERT INTO JOB_HISTORY
                (EMPLOYEE_ID,
                 JOB_CODE,
                 JOB_START,
                 JOB_END,
                 DEPARTMENT_CODE,
                 SUPERVISOR_ID)
        VALUES  (:EMPL-ID,
                 :JOB-CODE,
                 :START-DATE,
                 :END-DATE,
                 :DEPT-CODE,
                 :SUPER)
END-EXEC

IF SQLCODE < 0 THEN
        EXEC SQL        ROLLBACK        END-EXEC
        DISPLAY "An error has occurred. Try again."
        GO TO STORE-JOB-HISTORY
END-IF

EXEC SQL        COMMIT  END-EXEC
```

Example 3: Copying from one table to another

This interactive SQL example copies a subset of data from the EMPLOYEES table to an identical intermediate result table. To do this, it uses a select expression that limits rows of the select expression's result table to those with data on employees who live in New Hampshire.

```
SQL> INSERT INTO TEMP
cont>     (EMPLOYEE_ID,
cont>      LAST_NAME,
cont>      FIRST_NAME,
cont>      MIDDLE_INITIAL,
cont>      ADDRESS_DATA_1,
cont>      ADDRESS_DATA_2,
cont>      CITY,
cont>      STATE,
cont>      POSTAL_CODE,
cont>      SEX,
cont>      BIRTHDAY,
Cont>      STATUS_CODE)
cont> SELECT * FROM EMPLOYEES
cont>   WHERE STATE = 'NH';
90 rows inserted
SQL>
```

Example 4: Copying rows between databases with the INSERT statement

This example copies the contents of the EMPLOYEES table from the personnel database to another database, LOCALDATA.

```
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> ATTACH 'ALIAS LOCALDB FILENAME localdata';
SQL>
SQL> DECLARE TRANSACTION
cont>   ON PERS USING (READ ONLY
cont>   RESERVING PERS.EMPLOYEES FOR SHARED READ)
cont> AND
cont>   ON LOCALDB USING (READ WRITE
cont>   RESERVING LOCALDB.EMPLOYEES FOR SHARED WRITE);
SQL>
SQL> INSERT INTO LOCALDB.EMPLOYEES
cont>   SELECT * FROM PERS.EMPLOYEES;
100 rows inserted
SQL>
```

Example 5: Adding data to columns of data type LIST OF BYTE VARYING

The following interactive SQL example adds a new row to the RESUMES table of the sample personnel database. It first assigns a value to the EMPLOYEE_ ID column, then adds three lines of information to the RESUME column of the same row. The RESUME column has the LIST OF BYTE VARYING data type. You must specify the name of the list column (RESUME) in addition to the table column when declaring the table cursor for a positioned insert.

```
SQL> DECLARE TBLCURSOR INSERT ONLY TABLE CURSOR FOR SELECT EMPLOYEE_ID, RESUME
cont> FROM RESUMES;
SQL> DECLARE LSTCURSOR INSERT ONLY LIST CURSOR FOR SELECT RESUME
cont> WHERE CURRENT OF TBLCURSOR;
SQL> OPEN TBLCURSOR;
SQL> INSERT INTO CURSOR TBLCURSOR (EMPLOYEE_ID) VALUES ('00167');
1 row inserted
SQL> OPEN LSTCURSOR;
SQL> INSERT INTO CURSOR LSTCURSOR VALUES ('This is the resume for 00167');
SQL> INSERT INTO CURSOR LSTCURSOR VALUES ('Boston, MA');
SQL> INSERT INTO CURSOR LSTCURSOR VALUES ('Oracle Corporation');
SQL> CLOSE LSTCURSOR;
SQL> CLOSE TBLCURSOR;
SQL> COMMIT;
```

Example 6: Using the PLACEMENT ONLY RETURNING DBKEY clause of the INSERT statement

```
SQL> INSERT INTO EMPLOYEES
cont> (EMPLOYEE_ID, LAST_NAME, FIRST_NAME)
cont> VALUES
cont> ('5000', 'Parsons', 'Diane')
cont> PLACEMENT ONLY RETURNING DBKEY;
                  DBKEY
               56:34:-1
1 row allocated
SQL>
```

## INTEGRATE Statement

OpenVMS VAX OpenVMS Alpha

Makes definitions in a database and in a repository correspond by changing definitions in either the database or the repository.

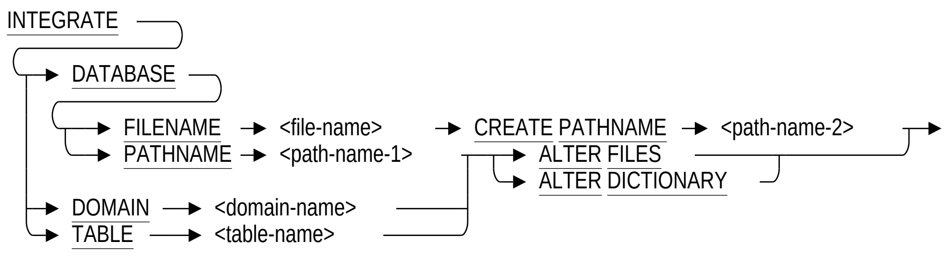
The INTEGRATE statement can also create database definitions in the repository by copying from a database file to a specified repository.

This statement can be used only on OpenVMS platforms.

### Environment

You can issue the INTEGRATE statement only in interactive SQL.

### Format



domain-name =

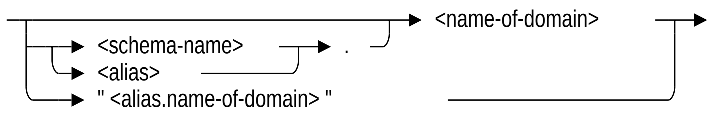
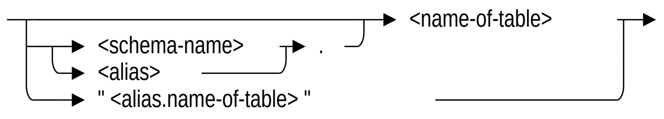


table-name =



## Arguments

**INTEGRATE DATABASE FILENAME file-name CREATE PATHNAME path-name-2**
Stores existing database system file definitions in the repository for the first time. See Example 7–3. Use the INTEGRATE DATABASE FILENAME clause if you did not specify PATHNAME or the repository was not installed when you created the database.

If you use the INTEGRATE DATABASE FILENAME clause, the repository database node specified in the path name must not exist. If older repository definitions do exist with the path name you are specifying, specify a different repository path name, placing the new database definitions elsewhere.

The file-name clause is the full or partial file specification that specifies the source of the database definitions. You do not need to specify the file extension. The database system automatically uses the database root file ending with the .rdb file extension.

Path-name-2 is the repository path name for the repository where the INTEGRATE statement creates the database definitions (using the database system files as the source). You can specify either a full repository path name or a relative repository path name. This must be the path name, not the name of the database itself.

**INTEGRATE DATABASE PATHNAME path-name-1 ALTER FILES**
Alters any table and domain definitions created with the CREATE TABLE FROM statement or the CREATE DOMAIN FROM statement so they match their sources in the repository. The INTEGRATE . . . ALTER FILES statement has no effect on definitions not created with the FROM clause. This is useful if the database file definitions no longer match the definitions in the repository. See Example 7–1.

Path-name-1 is the repository path name for the repository database that is the source for altering the definitions in the database. You can specify either a full repository path name or a relative repository path name.

**Caution**

Using the ALTER FILES clause may destroy data associated with definitions in your database file if those definitions are not defined in your repository. In this situation, you will lose real data. For this reason, use the ALTER FILES clause with caution.

**INTEGRATE DATABASE PATHNAME path-name-1 ALTER DICTIONARY**
Alters the database definitions in the dictionary so they are the same as those in the database. This is useful if repository definitions no longer match the definitions in the database file. See Example 7–2. Note, though, that altering the database definitions may affect other applications that refer to these definitions.

The repository must already exist and may contain definitions.

Path-name-1 is the repository path name for the repository database that SQL alters using the definitions in the database file as a source. You can specify either a full repository path name or a relative path name.

**INTEGRATE DOMAIN domain-name ALTER FILES**
Alters the domain definitions in the database to match the field definitions in the repository. Collating sequences referenced by the domain and columns that are based on the domain and the tables that contain them may also be altered if they have changed in the repository.

**INTEGRATE DOMAIN domain-name ALTER DICTIONARY**
Alters the field definitions in the repository to match the domain definitions in the database. Collating sequences referenced by the domain and columns that are based on the domain and the tables that contain them may also be altered if they have changed in the database.

**INTEGRATE TABLE table-name ALTER FILES**
Alters the table definitions in the database to match the record definitions in the repository. Other objects referencing the table or that are referenced by it and have changed definition in the repository may be altered. These other objects are:

- Domains
- Collating sequences
- Other referenced tables and columns
- Foreign key constraints and check constraints
- Indexes
- Views that reference the table
- Storage maps and storage areas referenced by an index

**INTEGRATE TABLE table-name ALTER DICTIONARY**
Alters the record definitions in the repository to match the table definitions in the database. Other objects referencing the table or that are referenced by it and have changed definitions in the database may be altered. These other objects are:

- Fields
- Collating sequences
- Other referenced records and fields
- Foreign key constraints and check constraints
- Indexes

## Usage Notes

- Do not issue the INTEGRATE statement when there are active users updating the database. Before entering the INTEGRATE statement, use the RMU Show Users command to make sure no other users are updating the database definitions.
- You must commit the transaction after entering the INTEGRATE statement.
- The INTEGRATE DATABASE statement implicitly attaches to the database.
- When using the INTEGRATE DOMAIN and INTEGRATE TABLE statements, you must attach by path name or define SQL$DATABASE to be the database from which or to which you want to integrate domains and tables.
- The domain or table specified in the INTEGRATE DOMAIN or the INTEGRATE TABLE statements must exist in both the repository and the database before it can be integrated. An error is returned if the named domain or table does not exist.
- The domain name or table name specified in the INTEGRATE DOMAIN ALTER DICTIONARY or the INTEGRATE TABLE ALTER DICTIONARY statements are not CDD/Repository path names but valid Oracle Rdb domain and table names.

## Examples

Example 7–1 shows how to use the INTEGRATE statement with the ALTER FILES clause. In this example, fields (domains) are defined in the repository. Then, using SQL, a table is created based on the repository definitions. Subsequently, the repository definitions are changed so the definitions in the database file and the repository no longer match. The INTEGRATE statement resolves this situation by altering the database definitions using the repository definitions as the source.

**Example 7–1 Updating the Database File Using Repository Definitions**

```
$ !
$ ! Define CDD$DEFAULT
$ !
$ DEFINE CDD$DEFAULT SYS$COMMON:[REPOSITORY]CATALOG
$ !
$ ! Enter the CDO to create new field and record definitions:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Create two field (domain) definitions in the repository:
CDO> !
CDO> DEFINE FIELD PART_NUMBER DATATYPE IS WORD.
CDO> DEFINE FIELD PRICE DATATYPE IS WORD.
CDO> !
CDO> ! Define a record called INVENTORY using the two
CDO> ! fields previously defined:
CDO> !
CDO> DEFINE RECORD INVENTORY.
CDO> PART_NUMBER.
CDO> PRICE.
CDO> END RECORD INVENTORY.
CDO> !
CDO> EXIT
```

(continued on next page)

**Example 7–1 (Cont.) Updating the Database File Using Repository Definitions**

```
$ !
$ ! Enter SQL:
$ !
$ SQL
SQL> !
SQL> ! In SQL, create the database ORDERS:
SQL> !
SQL> CREATE DATABASE ALIAS ORDERS PATHNAME ORDERS;
SQL> !
SQL> ! Create a table in the database ORDERS using the
SQL> ! INVENTORY record (table) just created in the repository:
SQL> !
SQL> CREATE TABLE FROM SYS$COMMON:[REPOSITORY]CATALOG.INVENTORY
cont> ALIAS ORDERS;
SQL> !
SQL> ! Use the SHOW TABLE statement to see information about
SQL> ! INVENTORY the table:
SQL> !
SQL> SHOW TABLE ORDERS.INVENTORY
Information for table ORDERS.INVENTORY

CDD Pathname:   SYS$COMMON:[REPOSITORY]CATALOG.INVENTORY;1

Columns for table ORDERS.INVENTORY:
Column Name                     Data Type        Domain
-----------                     ---------        ------
PART_NUMBER                     SMALLINT         ORDERS.PART_NUMBER
PRICE                           SMALLINT         ORDERS.PRICE
   .
   .
   .
SQL> COMMIT;
SQL> EXIT
```

(continued on next page)

**Example 7–1 (Cont.) Updating the Database File Using Repository Definitions**

```
$ !
$ ! Enter CDO again:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Verify that the fields PART_NUMBER and PRICE are
cdo> ! in the record INVENTORY:
CDO> !
CDO> SHOW RECORD INVENTORY
Definition of record INVENTORY
|   Contains field          PART_NUMBER
|   Contains field          PRICE
CDO> !
CDO> ! Define the fields VENDOR_NAME and QUANTITY. Add them to
CDO> ! the record INVENTORY using the CDO CHANGE RECORD command. Now, the
CDO> ! definitions used by the database no longer match the definitions
CDO> ! in the respository, as the CDO message indicates:
CDO> !
CDO> DEFINE FIELD VENDOR_NAME DATATYPE IS TEXT 20.
CDO> DEFINE FIELD QUANTITY DATATYPE IS WORD.
CDO> !
CDO> CHANGE RECORD INVENTORY.
CDO> DEFINE VENDOR_NAME.
CDO> END.
CDO> DEFINE QUANTITY.
CDO> END.
CDO> END INVENTORY RECORD.
%CDO-I-DBMBR, database SQL_USER:[PRODUCTION]CATALOG.ORDERS(1) may need
to be INTEGRATED
CDO> !
CDO> ! Use the SHOW RECORD command to see if the fields VENDOR_NAME
CDO> ! and QUANTITY are part of the INVENTORY record:
CDO> !
CDO> SHOW RECORD INVENTORY
Definition of record INVENTORY
|   Contains field          PART_NUMBER
|   Contains field          PRICE
|   Contains field          VENDOR_NAME
|   Contains field          QUANTITY
CDO> !
CDO> EXIT
```

(continued on next page)

**Example 7–1 (Cont.) Updating the Database File Using Repository Definitions**

```
$ !
$ ! Enter SQL again:
$ !
$ SQL
SQL> !
SQL> ! Use the INTEGRATE  . . .   ALTER FILES statement to update
SQL> ! the definitions in the database file, using the repository definitions
SQL> ! as the source. Note the INTEGRATE statement implicitly attaches to
SQL> ! the database.
SQL> !
SQL> INTEGRATE DATABASE PATHNAME SYS$COMMON:[REPOSITORY]CATALOG.ORDERS
cont> ALTER FILES;
SQL> !
SQL> ! Use the SHOW TABLE statement to see if the table INVENTORY has
SQL> ! changed.  SQL has added the VENDOR_NAME and QUANTITY domains
SQL> ! to the database file:
SQL> !
SQL> SHOW TABLE INVENTORY
Information for table INVENTORY

CDD Pathname:   SYS$COMMON:[REPOSITORY]CATALOG.INVENTORY;1

Columns for table INVENTORY:
Column Name                     Data Type        Domain
-----------                     ---------        ------
PART_NUMBER                     SMALLINT         PART_NUMBER
PRICE                           SMALLINT         PRICE
VENDOR_NAME                     CHAR(20)         VENDOR_NAME
QUANTITY                        SMALLINT         QUANTITY
   .
   .
   .
SQL> COMMIT;
SQL> EXIT
```

Example 7–2 shows how to update the repository using the database files as the source by issuing the INTEGRATE statement with the ALTER DICTIONARY clause. The example starts with the definitions in the repository matching the definitions in the database file. There is a table in the database and a record in the repository, both called CUSTOMER_ORDERS. The CUSTOMER_ORDERS table has four columns based on four domains of the same name: FIRST_ORDER, SECOND_ORDER, THIRD_ORDER, and FOURTH_ORDER.

This example adds to the database file a domain called FIFTH_DOM, on which the local column called FIFTH_ORDER is based. At this point, the database file and the repository definitions no longer match. The INTEGRATE . . . ALTER DICTIONARY statement resolves this situation by altering the repository using the database file definitions as the source.

**Example 7–2 Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause**

```
SQL> ! Create the database using the PATHNAME clause:
SQL> !
SQL> CREATE DATABASE FILENAME TEST1
cont>              PATHNAME SYS$COMMON:[REPOSITORY]TEST1;
SQL> !
SQL> ! Create domains for the TEST1 database:
SQL> !
SQL> CREATE DOMAIN FIRST_ORDER CHAR(4);
SQL> CREATE DOMAIN SECOND_ORDER CHAR(4);
SQL> CREATE DOMAIN THIRD_ORDER CHAR(4);
SQL> CREATE DOMAIN FOURTH_ORDER CHAR(4);
SQL> CREATE TABLE CUSTOMER_ORDERS
cont>     (FIRST_ORDER FIRST_ORDER,
cont>      SECOND_ORDER SECOND_ORDER,
cont>      THIRD_ORDER THIRD_ORDER,
cont>      FOURTH_ORDER FOURTH_ORDER);
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
SQL> !
SQL> ! Attach to the database with the FILENAME clause so the
SQL> ! repository is not updated:
SQL> !
SQL> ATTACH  'ALIAS TEST1 FILENAME TEST1';
SQL> !
SQL> ! Use the SHOW TABLE statement to see what columns and domains
SQL> ! are part of the table CUSTOMER_ORDERS:
SQL> !
SQL> SHOW TABLE (COLUMNS) TEST1.CUSTOMER_ORDERS;
Information on table TEST1.CUSTOMER_ORDERS

Columns for table TEST1.CUSTOMER_ORDERS:

Column Name                     Data Type        Domain
-----------                     ---------        ------
FIRST_ORDER                     CHAR(4)          FIRST_ORDER
SECOND_ORDER                    CHAR(4)          SECOND_ORDER
THIRD_ORDER                     CHAR(4)          THIRD_ORDER
FOURTH_ORDER                    CHAR(4)          FOURTH_ORDER
```

(continued on next page)

**Example 7–2 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause**

```
SQL> !
SQL> ! Create a new domain called FIFTH_DOM.  Add a new
SQL> ! column to the CUSTOMER_ORDERS table called FIFTH_ORDER
SQL> ! and base it on the domain FIFTH_DOM:
SQL> !
SQL> CREATE DOMAIN TEST1.FIFTH_DOM CHAR(4);
SQL> ALTER TABLE TEST1.CUSTOMER_ORDERS ADD FIFTH_ORDER TEST1.FIFTH_DOM;
SQL> !
SQL> ! Check the CUSTOMER_ORDERS table to verify that the column FIFTH_ORDER
SQL> ! was created:
SQL> !
SQL> SHOW TABLE (COLUMNS) TEST1.CUSTOMER_ORDERS;

Information on table TEST1.CUSTOMER_ORDERS

Column Name                     Data Type        Domain
-----------                     ---------        ------
FIRST_ORDER                     CHAR(4)          TEST1.FIRST_ORDER
SECOND_ORDER                    CHAR(4)          TEST1.SECOND_ORDER
THIRD_ORDER                     CHAR(4)          TEST1.THIRD_ORDER
FOURTH_ORDER                    CHAR(4)          TEST1.FOURTH_ORDER
FIFTH_ORDER                     CHAR(4)          TEST1.FIFTH_DOM
SQL> COMMIT;
SQL> EXIT
$ !
$ ! Invoke CDO:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
```

(continued on next page)

**Example 7–2 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause**

```
CDO> !
CDO> ! Note that only the database definition for TEST1 appears in the
CDO> ! repository directory:
CDO> !
  DIRECTORY
Directory SYS$COMMON:[REPOSITORY]
TEST1(1)                                   CDD$DATABASE
CDO> !
CDO> ! Check the record CUSTOMER_ORDERS.  The field FIFTH_ORDER is not part of
CDO> ! the record CUSTOMER_ORDERS.  This means that the definitions in the
CDO> ! database file do not match the definitions in the repository.
CDO> !
CDO> !
CDO> SHOW RECORD CUSTOMER_ORDERS FROM DATABASE TEST1
Definition of the record CUSTOMER_ORDERS
|   Contains field              FIRST_ORDER
|   Contains field              SECOND_ORDER
|   Contains field              THIRD_ORDER
|   Contains field              FOURTH_ORDER
CDO> EXIT
$ !
$ ! Enter SQL again:
$ !
$ SQL
SQL> !
SQL> ! To make the definitions in the repository match those in the database
SQL> ! file, use the INTEGRATE statement with the ALTER DICTIONARY clause.
SQL> ! Note that the INTEGRATE statement implicitly attaches to the
SQL> ! database.
SQL> !
SQL> INTEGRATE DATABASE PATHNAME TEST1 ALTER DICTIONARY;
SQL> COMMIT;
SQL> EXIT
$ !
$ ! Enter CDO again:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
```

(continued on next page)

**Example 7–2 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause**

```
CDO> ! Use the SHOW RECORD command to verify that the field FIFTH_ORDER is now
CDO> ! part of the record CUSTOMER_ORDERS.  Now, the definitions in both the
CDO> ! repository and the database file are the same.
CDO> !
CDO> SHOW RECORD CUSTOMER_ORDERS FROM DATABASE TEST1
Definition of record CUSTOMER_ORDERS
|   Contains field           FIRST_ORDER
|   Contains field           SECOND_ORDER
|   Contains field           THIRD_ORDER
|   Contains field           FOURTH_ORDER
|   Contains field           FIFTH_ORDER
CDO> !
CDO> ! Use the ENTER command to make the record (table) CUSTOMER_ORDERS and
CDO> ! its fields (domains) appear in the repository.  The ENTER command
CDO> ! assigns a repository directory name to an element.
CDO> !
CDO> ENTER FIELD FIRST_ORDER FROM DATABASE TEST1
CDO> !
CDO> ! Verify that a repository path name was assigned to the field
CDO> ! FIRST_ORDER:
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]
FIRST_ORDER(1)                          FIELD
TEST1(1)                                CDD$DATABASE
CDO> ENTER FIELD SECOND_ORDER FROM DATABASE TEST1
   .
   .
   .
CDO> ENTER FIELD FIFTH_DOM FROM DATABASE TEST1
CDO> !
CDO> ! Now all the domains and tables in TEST1 have been assigned a
CDO> ! repository directory name:
CDO> DIRECTORY
```

(continued on next page)

**Example 7–2 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause**

```
Directory SYS$COMMON:[REPOSITORY]
CUSTOMER_ORDERS(1)                       RECORD
FIFTH_DOM(1)                             FIELD
FIRST_ORDER(1)                           FIELD
FOURTH_ORDER(1)                          FIELD
SECOND_ORDER(1)                          FIELD
TEST1(1)                                 CDD$DATABASE
THIRD_ORDER(1)                           FIELD
```

To store existing database file definitions in the repository for the first time, use the INTEGRATE statement with the CREATE PATHNAME clause. This statement builds repository definitions using the database file as the source.

Example 7–3 shows how to store existing database system file definitions in the repository for the first time. This example first creates a database only in a database file, not in the repository. Next, the INTEGRATE statement with the CREATE PATHNAME clause updates the repository with the data definitions from the database system file.

**Example 7–3 Storing Existing Database File Definitions in the Repository**

```
SQL> !
SQL> ! Create a database without requiring the repository (the default)
SQL> ! or specifying a path name:
SQL> !
SQL> CREATE DATABASE ALIAS DOGS;
SQL> !
SQL> ! Now create a table for the breed of dog, poodles.  The
SQL> ! columns in the table are types of poodles:
SQL> !
SQL> CREATE TABLE DOGS.POODLES
cont> ( STANDARD  CHAR(10),
cont>   MINIATURE CHAR(10),
cont>   TOY       CHAR(10) );
```

(continued on next page)

**Example 7–3 (Cont.) Storing Existing Database File Definitions in the Repository**

```
SQL> !
SQL> ! Use the SHOW TABLE statement to see the table POODLES:
SQL> !
SQL> SHOW TABLE (COLUMNS) DOGS.POODLES
Information on table DOGS.POODLES

Columns for table DOGS.POODLES:
Column Name                     Data Type        Domain
-----------                     ---------        ------
STANDARD                        CHAR(10)
MINIATURE                       CHAR(10)
TOY                             CHAR(10)

SQL> COMMIT;
SQL> EXIT
$ !
$ ! Enter CDO:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Use the DIRECTORY command to check if the database definition DOGS is
CDO> ! in the repository:
CDO> !
CDO> DIRECTORY
Directory SYS$COMMON:[REPOSITORY]
%CDO-E-NOTFOUND, entity  not found in dictionary
CDO> !
CDO> ! DOGS is not in the repository.
CDO> !
CDO> EXIT
$ !
$ ! Enter SQL again:
$ !
$ SQL
SQL> !
SQL> ! Use the INTEGRATE statement using the CREATE PATHNAME clause to
SQL> ! update the repository using the DOGS database file:
SQL> !
SQL> INTEGRATE DATABASE FILENAME SQL_USER:[PRODUCTION.ANIMALS]DOGS
cont> CREATE PATHNAME SYS$COMMON:[REPOSITORY]DOGS;
SQL> COMMIT;
SQL> EXIT
```

(continued on next page)

**Example 7–3 (Cont.) Storing Existing Database File Definitions in the Repository**

```
$ !
$ ! Enter CDO again:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Use the DIRECTORY command to check if the database definition DOGS
CDO> ! has been integrated into the repository:
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]
DOGS(1)                                    CDD$DATABASE
CDO> !
CDO> ! You can also use the SHOW USED_BY command to see
CDO> ! if the record (table) POODLES and the fields (columns)
CDO> ! STANDARD, MINIATURE, and TOY are part of the database
CDO> ! definition DOGS.
CDO> !
```

(continued on next page)

**Example 7–3 (Cont.) Storing Existing Database File Definitions in the Repository**

```
CDO> SHOW USED_BY/FULL DOGS
Members of SYS$COMMON:[REPOSITORY]DOGS(1)
|   DOGS                            (Type : CDD$RDB_DATABASE)
|   |   via CDD$DATABASE_SCHEMA
   .
   .
   .
|   SYS$COMMON:[REPOSITORY]CDD$RDB_SYSTEM_METADATA.RDB$CDD_NAME;1(Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   POODLES                         (Type : RECORD)
|   |   |   via CDD$RDB_DATA_AGGREGATE
|   |   |   STANDARD                        (Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   |   |   SQL$10CHR                       (Type : FIELD)
|   |   |   |   |   via CDD$DATA_ELEMENT_BASED_ON
|   |   |   MINIATURE                       (Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   |   |   SQL$10CHR                       (Type : FIELD)
|   |   |   |   |   via CDD$DATA_ELEMENT_BASED_ON
|   |   |   TOY                             (Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   |   |   SQL$10CHR                       (Type : FIELD)
|   |   |   |   |   via CDD$DATA_ELEMENT_BASED_ON
   .
   .
   .
CDO> EXIT
```

Example 7–4 shows how to update a repository field using the database files as the source by issuing the INTEGRATE DOMAIN statement with the ALTER DICTIONARY clause. The example starts with the definitions in the repository matching the definitions in the database file. There is a domain in the database and a field in the repository, both called DOMTEST.

This example alters the domain in the database file name TESTDB. At this point, the database file and the repository definitions no longer match. The INTEGRATE DOMAIN . . . ALTER DICTIONARY statement resolves this situation by altering the repository using the database file definitions as the source.

**Example 7–4 Modifying Repository Field Using the INTEGRATE DOMAIN Statement with the ALTER DICTIONARY Clause**

```
SQL> -- Create a database, domain, and table.
SQL> --
SQL> CREATE DATABASE FILENAME TESTDB PATHNAME TESTDB;
SQL> CREATE COLLATING SEQUENCE FRENCH FRENCH;
SQL> CREATE DOMAIN DOMTEST
cont>    CHAR(5)
cont>    COLLATING SEQUENCE IS FRENCH;
SQL> CREATE DOMAIN TEST_DOM_1
cont>    CHAR(1);
SQL> CREATE TABLE TEMP_TAB
cont>    (ROW1 CHAR(5),
cont>     ROW2 DOMTEST,
cont>     ROW3 TEST_DOM_1,
cont>     ROW4 INT);
SQL> COMMIT;
SQL> SHOW DOMAIN DOMTEST
DOMTEST                        CHAR(5)
 Collating sequence: FRENCH
SQL> --
SQL> -- Disconnect from the database and invoke CDD/Repository V6.1
SQL> -- user interface and show the field DOMTEST from the TESTDB
SQL> -- database.
SQL> --
SQL> DISCONNECT ALL;
SQL> EXIT
$ CDO
Welcome to CDO V2.3
The CDD/Repository V6.1 User Interface
Type HELP for help
CDO> SHOW FIELD DOMTEST FROM DATABASE TESTDB
Definition of field DOMTEST
|   Datatype                text size is 5 characters
|   Collating sequence      'FRENCH'
```

(continued on next page)

**Example 7–4 (Cont.) Modifying Repository Field Using the INTEGRATE DOMAIN Statement with the ALTER DICTIONARY Clause**

```
CDO> !
CDO> ! Exit from CDD/Repository and attach to the database by file name
CDO> ! only.
CDO> !
CDO> EXIT
SQL> ATTACH 'FILENAME TESTDB';
SQL> --
SQL> -- Alter the domain DOMTEST.
SQL> --
SQL> ALTER DOMAIN DOMTEST
cont>    CHAR(10)
cont>    COLLATING SEQUENCE IS FRENCH;
SQL> COMMIT;
SQL> SHOW DOMAIN DOMTEST
DOMTEST                         CHAR(10)
 Collating sequence: FRENCH
SQL> --
SQL> -- Disconnect from the database and attach by path name only to issue
SQL> -- the INTEGRATE DOMAIN statement.
SQL> --
SQL> DISCONNECT ALL;
SQL> ATTACH 'PATHNAME TESTDB';
SQL> INTEGRATE DOMAIN DOMTEST ALTER DICTIONARY;
SQL> COMMIT;
SQL> --
SQL> -- Disconnect from the database and invoke CDD/Repository V6.1
SQL> -- user interface and show the altered field DOMTEST from the TESTDB
SQL> -- database.
SQL> --
SQL> DISCONNECT ALL;
SQL> EXIT
```

(continued on next page)

**Example 7–4 (Cont.) Modifying Repository Field Using the INTEGRATE DOMAIN Statement with the ALTER DICTIONARY Clause**

```
$ CDO
Welcome to CDO V2.3
The CDD/Repository V6.1 User Interface
Type HELP for help
CDO> SHOW FIELD DOMTEST FROM DATABASE TESTDB
Definition of field DOMTEST
|   Datatype                 text size is 10 characters
|   Collating sequence       'FRENCH'
|   Generic CDD$DATA_ELEMENT_CHARSET is         '0'
```

♦

## LEAVE Control Statement

Unconditionally ends execution within a compound statement block or a LOOP statement but resumes execution on any SQL statement that immediately follows the exited statement.

You can use the LEAVE statement within a labeled compound or LOOP statement only.

### Environment

You can use the LEAVE control statement in a compound statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
leave-statement =

──> LEAVE <statement-label>        ──>
```

### Arguments

**statement-label**
Names the label assigned to a compound statement or a LOOP statement in a multistatement procedure block. The LEAVE statement must specify the beginning label of the compound statement or the LOOP statement in which the LEAVE statement resides.

### Usage Notes

The LEAVE statement can specify the name of the procedure if the compound statement it contains is not labeled. See the Compound Statement.

## Examples

Example 1: Using the LEAVE control statement within a loop

```
-- This procedure counts the employees of a given state who have had
-- a decrease in their salary during their employement.

PROCEDURE COUNT_DECREASED (SQLSTATE :state CHAR(2),
                                    :n_decreased INTEGER);

BEGIN
   DECLARE :last_salary INTEGER (2);
   SET n_decreased = 0;

   emp_loop:
    FOR :empfor
       AS EACH ROW OF
          SELECT * FROM EMPLOYEES emp WHERE STATE = :state
          DO
          SET :last_salary = 0;

           history_loop:
           FOR :salfor
              AS EACH ROW OF
                SELECT salary_amount FROM  salary_history s
                        WHERE s.EMPLOYEE_ID = :empfor.EMPLOYEE_ID
           DO
             IF SALARY_AMOUNT > :LAST_SALARY
               THEN

                SET :N_DECREASED = :N_DECREASED +1 ;
                LEAVE HISTORY_LOOP;
             END IF;

             SET :LAST_SALARY = :SALARY_AMOUNT;
           END FOR;
   END FOR;
END;
```

## LOOP Control Statement

Allows the repetitive execution of one or more SQL statements in a compound statement. You can use the LOOP control statement in two ways:

- Loop until an error exception occurs or a LEAVE statement is executed
- Loop as long as a WHILE predicate clause is evaluated as TRUE

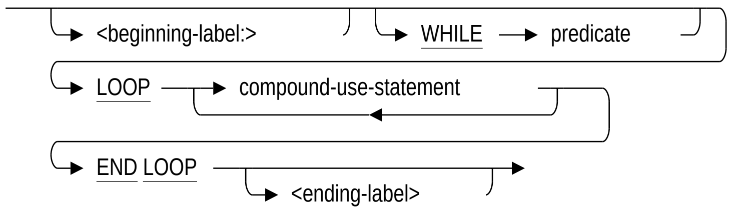
### Environment

You can use the LOOP control statement only within a compound statement:

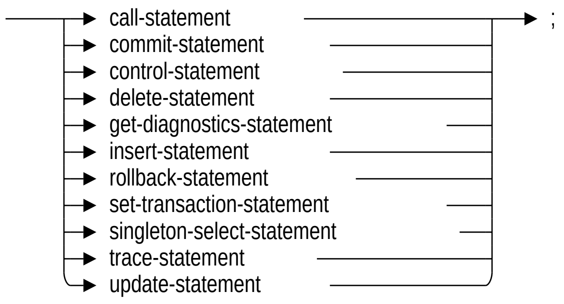
- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

loop-statement =



compound-use-statement =



## Arguments

**beginning-label:**
Assigns a name to a control loop. A beginning label used with the LEAVE statement lets you perform a controlled exit from a loop. A named loop is called a **labeled loop statement**. If you include an ending label, it must be identical to its corresponding beginning label. A beginning label must be unique within the procedure in which the label is contained.

**WHILE predicate**
Specifies a search condition that controls how many times SQL can execute a compound statement.

SQL evaluates the WHILE search condition. If it evaluates to TRUE, SQL executes the associated sequence of SQL statements (called a compound statement). If SQL does not encounter an error exception, control returns to the WHILE clause at the top of the loop for subsequent evaluation. Each time the search condition evaluates to TRUE, the WHILE-LOOP statement executes the SQL statements embedded within its LOOP . . . END LOOP block. If the search condition evaluates to FALSE or UNKNOWN, SQL bypasses the LOOP . . . END LOOP block and passes control to the statement after the LOOP statement.

**LOOP**
Marks the start of a control loop. A LOOP statement enables you to execute the associated sequence of SQL statements called a compound statement. After SQL executes the statements within the loop, control returns to the LOOP statement at the top of the loop for subsequent statement execution. Looping occurs until SQL encounters an error exception or executes a LEAVE statement. In either case, SQL passes control out of the LOOP block to the statement immediately after the LOOP statement.

**compound-use-statement**
Identifies the SQL statements allowed in a compound statement block. See the Compound Statement for the list of valid statements.

**END LOOP ending-label**
Marks the end of a control loop. If you choose to include the optional ending label, it must match exactly its corresponding beginning label. An ending label must be unique within the procedure in which the label is contained.

The optional end-label argument makes multistatement procedures easier to read, especially in very complex multistatement procedure blocks.

## Usage Notes

When you specify a WHILE clause be sure that you assign a new value to the search condition after each loop execution; otherwise, you create an infinite loop.

## Examples

Example 1: Executing a loop statement controlled by a WHILE search condition

The following example enters 5 rows for the student Robert Jones and uses the CAST function to insert placeholders for classes:

```
EXEC SQL
    BEGIN
        DECLARE :n INTEGER DEFAULT 100;
        WHILE :n > 0
        LOOP
            INSERT INTO enrollments
                VALUES ('Jones', 'Robert', '
                        'Class ' || CAST (:n AS CHAR (1));
            SET :n  = :n - 1;
        END LOOP;
    END;
```

## OPEN Statement

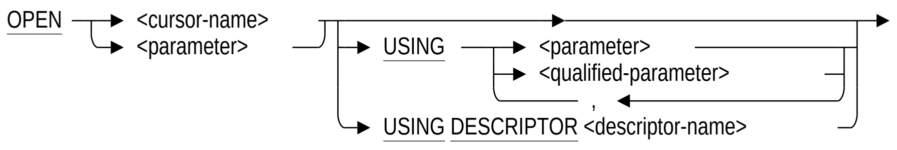
Opens a cursor so that rows of its result table can be retrieved through FETCH statements. The OPEN statement places the cursor before the first row of its result table.

### Environment

You can use the OPEN statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

### Format



### Arguments

**cursor-name**
**parameter**
Specifies the name of the cursor you want to open. Use a parameter if the cursor referred to by the cursor name was declared at run time with a dynamic DECLARE CURSOR statement. Specify the parameter used for the cursor name in the extended dynamic DECLARE CURSOR statement.

You can use a parameter to refer to the cursor name only when the OPEN statement refers to an extended dynamic cursor.

**USING parameter**
**USING qualified-parameter**
**USING DESCRIPTOR descriptor-name**
Specifies in dynamic SQL parameters (host language variables in a precompiled OPEN statement or formal parameters in an OPEN statement that is part of an SQL module language procedure) or qualified parameters (structures) whose values SQL uses to replace parameter markers in a prepared SELECT

statement named in the cursor declaration. These parameters are not for use in interactive SQL. SQL replaces the parameter markers with the values of the host language variables when it evaluates the SELECT statement of the cursor. See Chapter 3 and Chapter 4 for more information on the SQL module language and the SQL precompiler, respectively.

You must specify the USING clause when both of the following conditions exist:

- The declaration of the cursor you are opening specifies a prepared SELECT statement name.
- The statement string for the prepared SELECT statement includes parameter markers.

SQL does not allow the USING clause in an OPEN statement for a cursor that is not based on a prepared SELECT statement. For more information on parameter markers, see the PREPARE Statement, and the chapter on dynamic SQL in the *Oracle Rdb7 Guide to SQL Programming*.

There are two ways to specify parameters in a USING clause:

- With a list of parameters. The number of parameters in the list must be the same as the number of parameter markers in the prepared SELECT statement. (If any of the parameters in an OPEN statement is a host structure, SQL counts the number of variables in that structure when it compares the number of parameters in the USING clause with the number of parameter markers in the prepared SELECT statement.)
- With the name of a descriptor that corresponds to an SQLDA. Specify the name of the descriptor in the USING DESCRIPTOR clause. If you use the INCLUDE statement to insert the SQLDA into your program, the descriptor name is simply SQLDA.

  The SQLDA is a collection of variables used only in dynamic SQL. In an OPEN statement, the SQLDA points to a number of host language variables with which SQL replaces the parameter markers in a prepared SELECT statement. The number of variables must match the number of parameter markers.

The data types of host language variables must be compatible with the values of the corresponding column of the cursor row.

## Usage Notes

- SQL does not restrict how many cursors you can have open at once. It is valid to declare and open more than one cursor at a time.
- An open table cursor can be positioned:
  - Before a row of its result table. When it executes an OPEN statement, SQL positions the cursor before the first row. When SQL executes a DELETE statement that refers to a cursor, SQL positions the cursor before the row immediately following the deleted row.
  - On a row of its result table (after a FETCH statement for any but the last row).
  - After the last row of its result table. When the cursor is positioned on the last row, any FETCH or DELETE statement from the cursor positions the cursor after the last row.
- You cannot open a cursor until it has been declared in a DECLARE CURSOR statement.
- If you issue an OPEN statement for a cursor that is already open, SQL generates an error message. The OPEN statement has no effect on the cursor.
- SQL evaluates any parameters in the select expression of a DECLARE CURSOR statement when it executes the OPEN statement for the cursor. SQL will not evaluate the parameters again until you close and then open the cursor again.
- An open list cursor can be positioned:
  - Before an element of a list. When it executes an OPEN statement, SQL positions the cursor before the first element.
  - On an element of the list (after a FETCH statement for any but the last element).
  - After the last element of its result table. When the cursor is positioned on the last element, any FETCH statement from the cursor positions the cursor after the last element.
- When you open a list cursor, the table cursor that provides the row context must be open and positioned on a row.

## Examples

OpenVMS VAX OpenVMS Alpha

Example 1: Opening a cursor declared in a PL/I program

This program fragment uses embedded DECLARE CURSOR, OPEN, and FETCH statements to retrieve and print the name and department of managers. The OPEN statement places the cursor at the beginning of rows to be fetched.

```
/* Declare the cursor: */
EXEC SQL DECLARE MANAGER CURSOR FOR
        SELECT E.FIRST_NAME, E.LAST_NAME, D.DEPARTMENT_NAME
                FROM EMPLOYEES E, DEPARTMENTS D
                WHERE E.EMPLOYEE_ID = D.MANAGER_ID ;

/* Open the cursor: */
EXEC SQL OPEN MANAGER;

/* Start a loop to process the rows of the cursor: */
DO WHILE (SQLCODE = 0);
        /* Retrieve the rows of the cursor
        and put the value in host language variables: */
        EXEC SQL FETCH MANAGER INTO :FNAME, :LNAME, :DNAME;
        /* Print the values in the variables: */
                        .
                        .
                        .
END;

/* Close the cursor: */
EXEC SQL CLOSE MANAGER;
```

♦

Example 2: Opening a cursor to insert list data

The following interactive SQL example uses cursors to add a new row to the RESUMES table of the sample personnel database:

```
SQL> DECLARE TBLCURSOR INSERT ONLY TABLE CURSOR FOR
cont> SELECT EMPLOYEE_ID, RESUME FROM RESUMES;
SQL> DECLARE LSTCURSOR INSERT ONLY LIST CURSOR FOR
cont> SELECT RESUME WHERE CURRENT OF TBLCURSOR;
SQL> OPEN TBLCURSOR;
SQL> INSERT INTO CURSOR TBLCURSOR (EMPLOYEE_ID)
cont> VALUES ("00167");
1 row inserted
```

```
SQL> OPEN LSTCURSOR;
SQL> INSERT INTO CURSOR LSTCURSOR
cont> VALUES ("This is the resume for 00167");
SQL> INSERT INTO CURSOR LSTCURSOR
cont> VALUES ("Boston, MA");
SQL> INSERT INTO CURSOR LSTCURSOR
cont> VALUES ("Oracle Corporation");
SQL> CLOSE LSTCURSOR;
SQL> CLOSE TBLCURSOR;
SQL> COMMIT;
```

## Operating System Invocation ($) Statement

Gives access to the operating system command line environment from within SQL.

The dollar sign ($) tells SQL to spawn a subprocess or child process and pass the rest of the line to the operating system for processing. You must follow the dollar sign with an operating system command. After the operating system processes the command, it logs out of the subprocess or child process and returns control to SQL.

### Environment

You can invoke operating system commands only in interactive SQL.

### Format

```
$ operating-system-command
```

### Arguments

**operating-system-command**
Specifies a valid operating system command.

### Usage Notes

- Because SQL spawns a subprocess or child process to execute the operating system command, you cannot use the dollar sign command to create logical names or configuration parameters that affect the current interactive session. For instance, you cannot use the dollar sign command to change the value of the SQL$DATABASE logical or the SQL_DATABASE configuration parameter.
- Interactive SQL interprets any command line that begins with a dollar sign ($) as the start of an operating system command line. This is true even if the dollar sign is a continuation of a string literal from the previous line, which can lead to confusing results.

```
SQL> INSERT INTO EMPLOYEES (CITY) VALUES("DollarSign -
cont> $City")
%DCL-W-IVVERB, unrecognized command verb - check validity and spelling
 \CITY");\
cont> ;
%SQL-F-UNTSTR, Unterminated string found
SQL>
```

## Examples

OpenVMS VAX OpenVMS Alpha

Example 1: Using the DCL DIRECTORY command from within SQL

```
SQL> $ DIRECTORY *.SQL

Directory DISK2:[DEPT3.ACCT]

DEFPRO.SQL;6    NOTEQUAL.SQL;1   QUERY.SQL;1    REFEXAM.SQL;12
STORE.SQL;1     UPDATE.SQL;2

Total of 6 files.
```

♦

Digital UNIX

Example 2: Using the Digital UNIX ls command from within SQL

```
SQL> $ ls *.rbf
mf_personnel.rbf
```

♦

## PREPARE Statement

Prepares an SQL statement dynamically generated by a program for execution, and assigns a name to that statement.

The PREPARE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not necessarily part of a program's source code, but can be generated while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

With the optional INTO clause, the PREPARE statement also writes information about any select list items in a prepared SELECT statement to the SQLDA. The SQLDA is a collection of variables used only in dynamic SQL programs. To use the SQLDA, host languages must support pointer variables that provide indirect access to storage by storing the address of data instead of directly storing data in the variable. The languages supported by the SQL precompiler that also support pointer variables are Ada, C, and PL/I. Any other language that supports pointer variables can use the SQLDA, but must call SQL module procedures that contain SQL statements instead of embedding the SQL statements directly in source code. The SQLDA provides information about dynamic SQL statements to the program and information about memory allocated by the program to SQL.

The PREPARE . . . INTO statement stores in the SQLDA the number and data types of any select list items of a prepared statement.

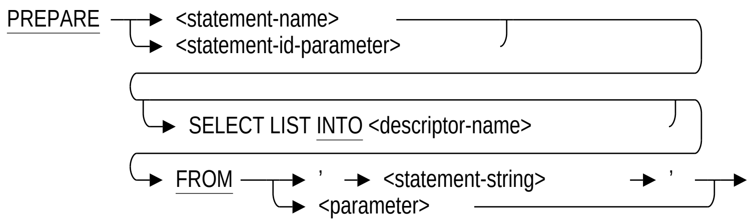
Appendix D describes in more detail the specific fields of the SQLDA, and how programs use it to communicate about select list items in prepared statements.

### Environment

You can use the PREPARE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

## Format



## Arguments

**statement-name**
**statement-id-parameter**
Identifies the prepared version of the SQL statement specified in the FROM clause. Depending on the type of SQL statement prepared, DESCRIBE, EXECUTE, and dynamic DECLARE CURSOR statements can refer to the statement name assigned in a PREPARE statement.

You can supply either a parameter or a compile-time statement name. Specifying a parameter lets SQL supply identifiers to programs at run time. Use an integer parameter to contain the statement identifier returned by SQL, or a character string parameter to contain the name of the statement that you pass to SQL. If you use the statement-id-parameter, and if that parameter is an integer, then you must explicitly initialize that integer to zero before executing the PREPARE statement.

A single set of dynamic SQL statements (PREPARE, DESCRIBE, EXECUTE, Extended Dynamic DECLARE CURSOR) can handle any number of dynamically executed statements. If you decide to use parameters, statements that refer to the prepared statement (DESCRIBE, EXECUTE, extended dynamic DECLARE CURSOR) must also use a parameter instead of the explicit statement name.

Refer to the DECLARE CURSOR Statement, Dynamic for an example demonstrating the PREPARE statement used with a dynamic DECLARE CURSOR statement.

**SELECT LIST INTO**
Specifies that SQL writes information about the number and data type of select list items in the statement string to the SQLDA. The SELECT LIST keywords clarify the effect of the INTO clause and are optional.

Using the SELECT LIST clause in a PREPARE statement is an alternative to issuing a separate DESCRIBE . . . SELECT LIST statement. See the DESCRIBE Statement for more information.

The SELECT LIST clause in a PREPARE statement is deprecated syntax. For more information about deprecated syntax, see Appendix F.

**Note**

The PREPARE statement LIST keyword is not related to the LIST data type or list cursors.

**descriptor-name**
Specifies the name of a structure declared in the host program as an SQLDA to which SQL writes information about select list items. Precompiled programs can use the embedded SQL statement INCLUDE SQLDA to automatically insert a declaration of an SQLDA structure, called SQLDA, in the program when it precompiles the program. Programs that use the SQL module language must explicitly declare an SQLDA. Either precompiled or SQL module language programs can explicitly declare additional SQLDAs, but must declare them with unique names. For sample declarations of SQLDA structures, see Section D.3.

**FROM statement-string**
**FROM parameter**
Specifies the SQL statement to be prepared for dynamic execution. You either specify the statement string directly enclosed in single quotation marks, or in a parameter (a host language variable in a precompiled PREPARE statement or a formal parameter in a PREPARE statement that is part of an SQL module language procedure) that contains the statement string.

Whether specified directly or by a parameter, the statement string must be a character string that is a dynamically executable SQL statement. (See the Usage Notes for a list of the SQL statements that can be dynamically executed.) If you specify the statement string directly, the maximum length is 1,024 characters. If you specify the statement string as a parameter, the maximum length of the statement string is 65,535 characters.

The form for the statement is the same as for embedded SQL statements, except that:

- You must not begin the string with EXEC SQL or end it with any statement terminator.

- In places where SQL allows host language variables in an embedded statement, you must specify parameter markers instead.

If you try to prepare an invalid statement, you will find a value in the SQLCODE, the SQLCODE field of the SQLCA, or the SQLSTATE status parameter indicating an error.

The values returned to the SQLCODE field are described in Appendix B. Check the message vector to see which error message was returned. If necessary, refer to the error message explanations and user actions located by default in SYS$HELP:SQL$MSG.DOC or /usr/lib/dbs/sql/vnn/help/sql_msg.doc.

**Parameter markers** are question marks (?) that denote parameters in the statement string of a PREPARE statement. Parameter markers are replaced by values in parameters or dynamic memory when the prepared statement is executed by an EXECUTE or OPEN statement.

## Usage Notes

- Some statements, such as INSERT and DELETE, return a count of the number of rows (on which the statement operated) in the SQLERRD[2] field of the SQLCA. To take advantage of this behavior, you must prepare the statement using the SQLCA as the status parameter. For more information about the SQLERRD[2] field, see Appendix B.
- You can execute the same prepared statement many times. However, if a statement to be dynamically executed does not contain select list items or parameter markers, and your program needs to execute it only once, you can use the EXECUTE IMMEDIATE statement to prepare and execute the statement in one step.
- The PREPARE . . . SELECT LIST form of the PREPARE statement, besides preparing a statement for execution, also stores information about the number and data type of select list items in the SQLDA. However, no form of the PREPARE statement corresponds to a DESCRIBE . . . MARKERS statement. To store information about parameter markers in the SQLDA, you must use the DESCRIBE . . . MARKERS statement.
- If you use the statement-id-parameter, and if that parameter is an integer, then you must explicitly intialize that integer to zero before executing the PREPARE statement.

- When you issue the EXECUTE statement for a previously prepared statement, you may be interested in obtaining information beyond the success or failure code returned in the SQLCODE status parameter. For example, you may want to know how many rows were affected by the execution of a DELETE or UPDATE statement. If you use an SQLCA status parameter, you can access this type of information.

  However, if you use an SQLCA parameter when you execute a prepared statement, you must first have used an SQLCA parameter when you prepared that statement. For example, using SQL module language calls from C, your code might look like the following where the SQLCA parameter is passed to both procedures:

  ```
  static struct SQLCA  sqlca;
  /* ... */
  PREPARE_STMT(&sqlca, statement, &stmt_id);
  /* ... */
  EXECUTE_STMT(&sqlca, &stmt_id);
  ```

- You cannot dynamically execute all statements that SQL allows you to embed in a precompiled program or make part of an SQL module language procedure. Statements you cannot dynamically execute are:

  - CLOSE
  - DECLARE CURSOR
  - DECLARE STATEMENT
  - DECLARE TABLE
  - DESCRIBE
  - EXECUTE
  - FETCH
  - INCLUDE
  - OPEN
  - PREPARE
  - RELEASE
  - WHENEVER

Digital UNIX

- Only the C, COBOL, FORTRAN, and Pascal languages are supported on Digital UNIX. ♦

Table 7–3 lists SQL statements that can be dynamically executed. It also shows whether the statements can have parameter markers or select list items that may have to be processed, and lists the associated nondynamic SQL statements used to process the statement dynamically.

**Table 7–3 SQL Statements That Can Be Dynamically Executed**

| Statement That Can Be Dynamically Executed | Parameter Markers Allowed? | Select List Items? | Associated Dynamic SQL Statements |
|---|---|---|---|
| SELECT (general form) | Yes | Yes | PREPARE<br>Dynamic DECLARE CURSOR<br>Extended dynamic DECLARE CURSOR<br>DESCRIBE (optional)<br>OPEN<br>FETCH<br>CLOSE<br>RELEASE (optional) |
| DELETE<br>INSERT<br>UPDATE | Yes | No | PREPARE<br>DESCRIBE (optional)<br>EXECUTE<br>EXECUTE IMMEDIATE (if no parameter markers)<br>RELEASE (optional) |

(continued on next page)

**Table 7–3 (Cont.) SQL Statements That Can Be Dynamically Executed**

| Statement That Can Be Dynamically Executed | Parameter Markers Allowed? | Select List Items? | Associated Dynamic SQL Statements |
|---|---|---|---|
| Compound statement<br>SELECT . . . INTO | Yes | Yes | PREPARE<br>DESCRIBE (optional)<br>EXECUTE<br>EXECUTE IMMEDIATE (if no parameter markers)<br>RELEASE (optional) |
| ALTER<br>ATTACH<br>DECLARE TRANSACTION<br>CREATE<br>COMMENT ON<br>COMMIT<br>DROP<br>GRANT<br>REVOKE<br>ROLLBACK<br>SET TRANSACTION | No | No | PREPARE<br>EXECUTE<br>EXECUTE IMMEDIATE<br>RELEASE (optional) |

## Example

OpenVMS VAX OpenVMS Alpha

Example 1: Preparing an INSERT statement with parameter markers

This PL/I program illustrates using a PREPARE statement to prepare an INSERT statement for dynamic execution. Because the statement string stored in COMMAND_STRING has parameter markers, the program needs to assign values to host language variables that will be substituted for the parameter markers during dynamic execution.

In this case, a DESCRIBE statement writes information about the parameter markers to the SQLDA and the program writes the addresses of the variables to the SQLDA. The program stores values in the variables and an EXECUTE statement substitutes the values for the parameter markers in the INSERT statement using the addresses in the SQLDA.

To shorten the example, this program is simplified:

- The program includes the INSERT statement as part of the program source code. A program with such coded SQL statements does not need to use dynamic SQL at all, but can simply embed the INSERT statement directly in the program. A program that must process SQL statements generated as it executes is the only type of program that requires dynamic SQL.
- The program declares host language variables for the parameter markers without first checking the SQLDA for their description. Typically, an application needs to look in the SQLDA to determine the number and data type of parameter markers in the statement string before allocating memory for them.

```
PREP_INTO: procedure options(main);
/*
*       Illustrate a dynamic INSERT statement
*       with parameter markers:
*/
declare FILESPEC char(20),
        EMP_ID CHAR(5),
        FNAME CHAR(10),
        LNAME CHAR(14),
        CITY CHAR(20),
        COMMAND_STRING char(256);

/* Declare communication area (SQLCA)
 * and descriptor area (SQLDA): */
EXEC SQL        INCLUDE SQLDA;
EXEC SQL        INCLUDE SQLCA;

/* Declare the database: */
EXEC SQL DECLARE SCHEMA RUNTIME FILENAME :FILESPEC;

/*
 *
 * procedure division
 *
 */

/*
 * Assign values to FILESPEC and COMMAND_STRING,
 * and allocate memory for the SQLDA:
 */
FILESPEC = 'SQL$DATABASE';
COMMAND_STRING =
        'INSERT INTO EMPLOYEES
        (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, CITY)
        VALUES (?,?,?,?)';
SQLSIZE = 10;
ALLOCATE SQLDA SET (SQLDAPTR);
SQLN = 10;
```

```
/*
 * Prepare the statement assigned to COMMAND_STRING:
 */
EXEC SQL PREPARE STMT3
        FROM COMMAND_STRING;

/* Use a DESCRIBE statement to write information
 * about the parameter markers in the statement string
 * to the SQLDA:
 */
EXEC SQL DESCRIBE STMT3 MARKERS INTO SQLDA;

/* Assign values to the variables: */
        EMP_ID = '99999';
        FNAME = 'Bob';
        LNAME = 'Addams';
        CITY = 'Francestown';

/*
 * Assign the addresses of the variables to the SQLDATA field
 * of the SQLDA:
 */
        SQLDATA(1) = ADDR(EMP_ID);
        SQLDATA(2) = ADDR(FNAME);
        SQLDATA(3) = ADDR(LNAME);
        SQLDATA(4) = ADDR(CITY);

/* Execute STMT3:*/
EXEC SQL EXECUTE STMT3 USING DESCRIPTOR SQLDA;

/*
* Display the contents of table S to make sure
* it has the proper contents and clean it up:
*/
CALL DUMP_S;
EXEC SQL DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = "99999";
EXEC SQL COMMIT WORK;
RETURN;

DUMP_S: PROC;
EXEC SQL DECLARE X CURSOR FOR SELECT
        EMPLOYEE_ID, FIRST_NAME, LAST_NAME, CITY
        FROM EMPLOYEES WHERE EMPLOYEE_ID = "99999";

/*
 * Declare a structure to hold values of rows from table S:
*/
DCL 1 S,
        2 EMP_ID CHAR(5),
        2 FNAME CHAR(10),
        2 LNAME CHAR(14),
        2 CITY CHAR(20);

/* Declare indicator vector for the preceding structure: */
DCL S_IND (4) FIXED(15) BIN;
```

```
PUT EDIT ('Dump the contents of S') (SKIP, SKIP, A);
EXEC SQL OPEN X;
EXEC SQL FETCH X INTO :S:S_IND;
DO WHILE (SQLCODE = 0);
   PUT EDIT (S_IND(1), ' ', S.EMP_ID, ' ')   (SKIP, F(6), A, A, A);
   PUT EDIT (S_IND(2), ' ', S.FNAME, ' ')  (F(6), A, A, A);
   PUT EDIT (S_IND(3), ' ', S.LNAME, ' ') (F(6), A, A, A);
   PUT EDIT (S_IND(4), ' ', S.CITY)   (F(6), A, A);
   EXEC SQL FETCH X INTO :S:S_IND;
END;
EXEC SQL CLOSE X;
RETURN;
END DUMP_S;

END PREP_INTO;
```

♦

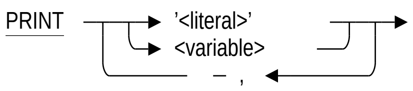
## PRINT Statement

Displays a message on the terminal in interactive SQL.

### Environment

You can use the PRINT statement in interactive SQL.

### Format



### Arguments

**'literal'**
Specifies the characters you want displayed to the user during execution of the command procedure. Enclose the characters in each literal within single quotation marks.

**variable**
Prints the definition of the specified variable.

### Usage Notes

- Use a comma to separate two or more literals. A comma used as a separator is not displayed to the user when the command procedure executes.
- To display a comma as part of a literal, include the comma inside the single quotation marks enclosing the literal.
- If you execute the PRINT statement within an SQL command procedure, SQL prints the output to SYS$OUTPUT. You cannot redefine SYS$OUTPUT to redirect output to a file. Use the SET OUTPUT statement to redirect the output to a file.

## Examples

Example 1: Displaying a literal from a command procedure

The following PRINT statement in a command procedure displays ’Creating trigger definitions for the database’ during the execution of the command procedure:

```
SQL> -- Trigger definition statements are next.
SQL> PRINT ’Creating trigger definitions for the database’;
SQL> CREATE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
   .
   .
   .
```

Example 2: Displaying a variable

The following PRINT statement displays the definition of a variable:

```
SQL> DECLARE :X CHAR(10);
SQL> BEGIN
cont>   SET :X = ’Active’;
cont> END;
SQL> PRINT :X;
 X
 Active
```

## QUIT Statement

Stops an interactive SQL session, rolls back any changes you made, and returns you to the DCL prompt.

### Environment

You can issue the QUIT statement in interactive SQL only.

### Format

```
QUIT
```

### Usage Notes

Both the QUIT and EXIT statements end an interactive SQL session. The QUIT statement automatically rolls back changes made during the session; the EXIT statement, by default, commits changes made during the session. The EXIT statement offers you a chance to roll back changes; QUIT does not offer a chance to commit changes.

## RELEASE Statement

Releases all resources used by a prepared dynamic SQL statement and prevents the prepared statement from executing again.

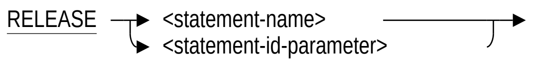
The RELEASE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not necessarily part of a program's source code, but can be generated while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

### Environment

You can use the RELEASE statement:

- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module

### Format



### Arguments

**statement-name**
**statement-id-parameter**
Specifies the name of a prepared statement or a statement name assigned in a PREPARE statement.

A single set of dynamic SQL statements (PREPARE, DESCRIBE, EXECUTE, dynamic DECLARE CURSOR) can handle any number of dynamically executed statements.

You can supply either a parameter or a compile-time statement name to identify the statement to be executed. Specifying a parameter lets SQL supply identifiers to programs at run time. Use an integer parameter to contain the statement identifier returned by SQL or a character string parameter to contain the name of the statement that you pass to SQL. If you use parameters, statements that refer to the prepared statement (DESCRIBE, EXECUTE, DECLARE CURSOR) must also use those parameters instead of the explicit statement name.

## Usage Notes

- When you prepare an SQL statement for dynamic execution, you cannot delete any schema definitions (such as constraints, indexes, or tables) referred to directly or indirectly by the statement until you release the statement.

  The RELEASE statement gives you a way to explicitly release prepared statements. SQL also implicitly releases dynamic SQL statements in the following circumstances:

  – After an EXECUTE IMMEDIATE statement

  – When a PREPARE statement refers to an already-prepared statement name

  – After a DISCONNECT statement

  You do not need to release statements for which the PREPARE statement failed, to do so is a programming error.

- If you have a prepared statement that refers to a cursor that is destroyed by a release of its own statement, executing the prepared statement produces unpredictable results. For example:

  ```
  DECLARE A CURSOR FOR A_STMT;
  PREPARE A_STMT FROM 'SELECT * FROM T';
  PREPARE B_STMT FROM 'DELETE T WHERE CURRENT OF A';

  OPEN A;
  FETCH A;
  EXECUTE B_STMT;
  CLOSE A;

  RELEASE A_STMT;

  EXECUTE B_STMT;   <--- This produces unpredictable results.
  ```

## Example

Example 1: Using the RELEASE statement

The following fragment from a COBOL program shows using a RELEASE statement to release resources from a prepared SELECT statement:

```
        .
        .
        .
FETCHES.
        DISPLAY "Here’s the row we stored:"

        EXEC SQL PREPARE STMT FROM
        ’SELECT * FROM EMPLOYEES WHERE EMPLOYEE_ID = "99999"’
        END-EXEC
        EXEC SQL DECLARE C CURSOR FOR STMT END-EXEC

        EXEC SQL OPEN C END-EXEC
        .
        .
        .
        EXEC SQL FETCH C INTO
                :EMP_ID:EMP_ID_IND,
                :LNAME:LNAME_IND,
                :FNAME:FNAME_IND,
                :MID_INIT:MID_INIT_IND,
                :ADDR_1:ADDR_1_IND,
                :ADDR_2:ADDR_2_IND,
                :CITY:CITY_IND,
                :STATE:STATE_IND,
                :P_CODE:P_CODE_IND,
                :SEX:SEX_IND,
                :BDATE:BDATE_IND,
                :S_CODE:S_CODE_IND
        END-EXEC

        DISPLAY EMP_ID," ",
                FNAME," ",
                MID_INIT," ",
                LNAME," ",
                ADDR_1," ",
                ADDR_2," ",
                CITY," ",
                STATE," ",
                P_CODE," ",
                SEX," ",
                BDATE," ",
                S_CODE.
```

```
PERFORM CHECK
EXEC SQL CLOSE C END-EXEC.
PERFORM CHECK.
EXEC SQL RELEASE STMT END-EXEC.
PERFORM CHECK.
.
.
.
```

## RETURN Control Statement

Returns the value of the stored function.

### Environment

You can use the RETURN statement in a compound statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
RETURN ───► value-expr      ───►
```

### Arguments

**RETURN value-expr**
Returns the result of a stored function defined with the CREATE MODULE statement.

See Section 2.6 for more information on value expressions.

### Usage Notes

- If the RETURN statement is omitted or is never executed, an exception is raised at run time.
- The RETURN statement is effective with stored functions only and not with stored procedures.
- The RETURN statement is not to be confused with the RETURNS clause of the stored function definition.
- The RETURN statement is required syntax when defining a stored function.
- The value-expr must be assignment-compatible with the data type defined by the stored function RETURNS clause.

## Examples

Example 1: Specifying the RETURN statement in a stored function

```
SQL> CREATE MODULE utility_functions
cont>   LANGUAGE SQL
cont>   FUNCTION abs (IN :arg INTEGER) RETURNS INTEGER
cont>      COMMENT 'Returns the absolute value of an integer';
cont>   BEGIN
cont>      RETURN CASE
cont>             WHEN :arg < 0 THEN - :arg
cont>             ELSE :arg
cont>             END;
cont>   END;
   .
   .
   .
cont> END MODULE;
```

## REVOKE Statement

Removes privileges from or entirely deletes an entry to the Oracle Rdb access privilege set (called the **access control list**) for a database, table, column, module, or external routine. Each entry in an access control list (ACL) consists of an identifier and a list of privileges assigned to the identifier.

- Each identifier specifies a user or a set of users.
- The list of privileges specifies which operations that user or user group can perform on the database, table, column, module, or external routine.

When a user tries to perform an operation on a database, SQL reads the associated ACL from top to bottom, comparing the identifier of the user with each entry. As soon as SQL finds the first match, it grants the rights listed in that entry and stops the search. All identifiers that do not match a previous entry are compared with the subsequent entry, and if no match occurs, they receive the rights of ("fall through" to) the entry [*,*], if it exists. If no entry has the user identifier [*,*], then unmatched user identifiers are denied all access to the database, table, or column. For this reason, both the entries and their order in the list are important.

To create or add privileges to an entry to the Oracle Rdb access privilege set for a database, table, view, column, module, or external routine, see the GRANT Statement.

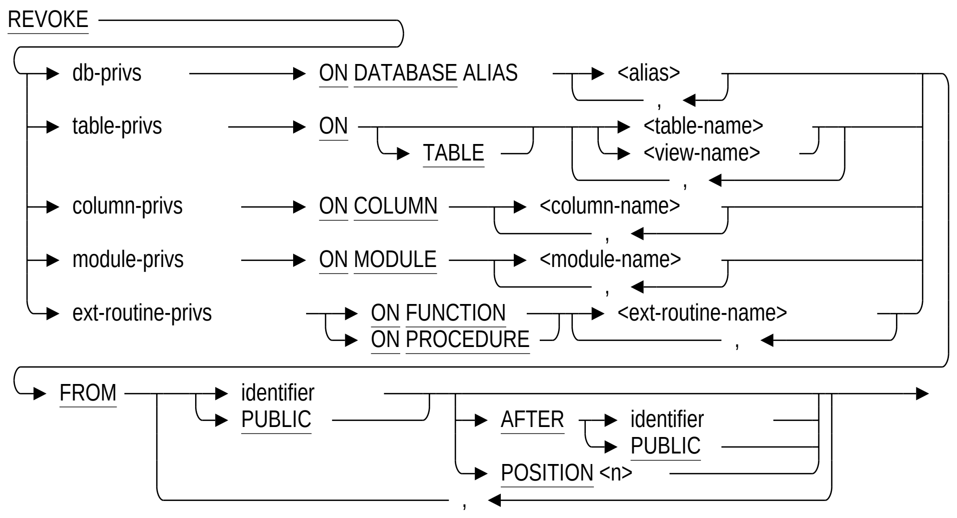
### Environment

You can use the REVOKE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

## REVOKE Statement

### Format



db-privs=

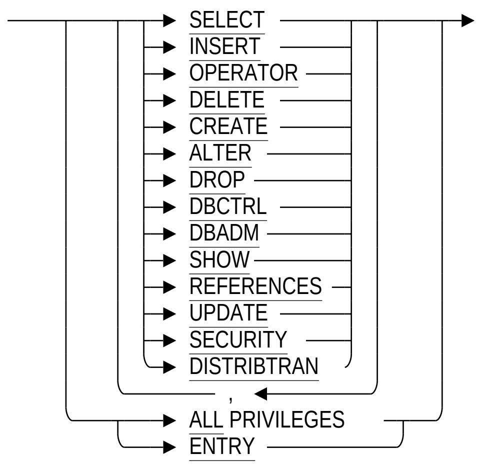
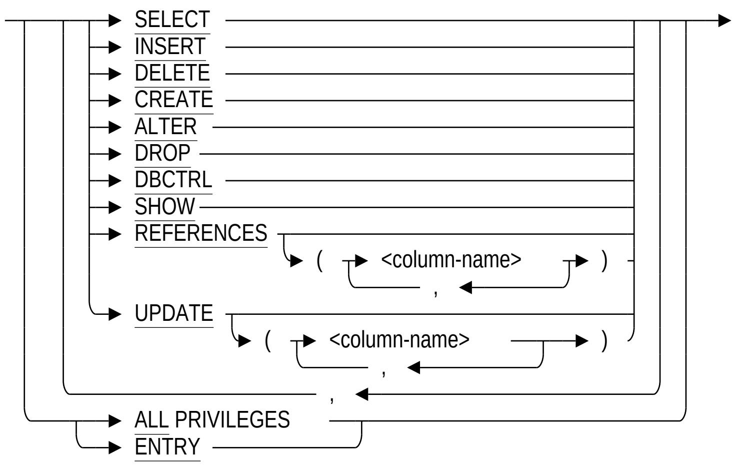
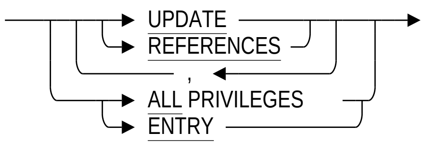


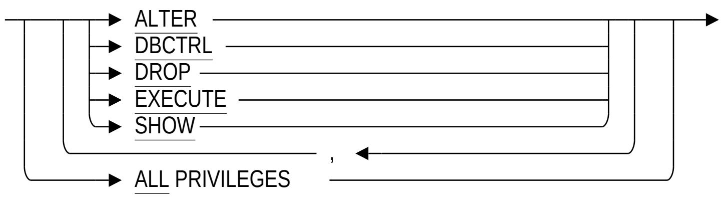
table-privs=



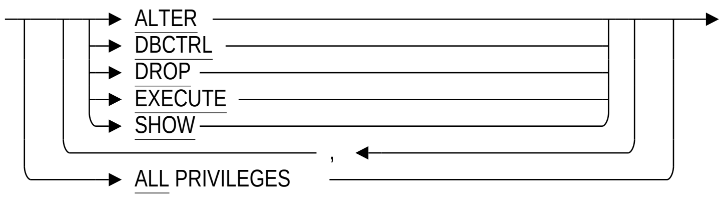
column-privs =



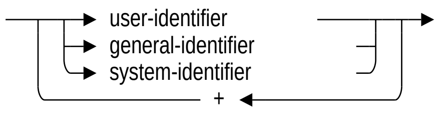
module-privs =



ext-routine-privs =



identifier =



## Arguments

**db-privs**
**table-privs**
**column-privs**
**module-privs**
**ext-routine-privs**

Specifies the list of privileges you want to remove from an existing ACL entry. The operations permitted by a given privilege keyword differ, depending on whether it was granted for a database, table, column, module, or external routine. Table 7–1 in the GRANT Statement lists the privilege keywords and their meanings for databases, tables, modules, and external routines.

You can only revoke column-level privileges that have been specifically granted at the column level.

For the SELECT, INSERT, and DELETE data manipulation privileges, SQL checks the ACL for the database and for the individual table before allowing access to a specific table. For example, if your SELECT privilege for a database that contains the EMPLOYEES table is revoked, you will not be able to read rows from the table even though you may have SELECT privilege to the EMPLOYEES table itself.

To revoke the data manipulation privileges UPDATE and REFERENCES, you must have at least read access to the database and the appropriate column privilege.

You cannot deny yourself the DBCTRL privilege for a database, table, module, or external routine that you create.

The SELECT privilege is a prerequisite for all other privileges. If you revoke the SELECT privilege, you effectively deny all privileges, even if they are specified in the privilege list. This restriction may cause REVOKE statements to fail when you might expect them to work. For instance, the following REVOKE statement fails because it tries to revoke the SELECT privilege from the ACL entry for the owner. Because that implicitly denies DBCTRL on the table to the owner, the statement fails.

```
SQL> REVOKE SELECT ON EMPLOYEES FROM serle;
%RDB-E-NO_PRIV, privilege denied by database facility
```

**ALL PRIVILEGES**
Specifies that SQL should revoke all privileges in the ACL entry. The REVOKE ALL PRIVILEGES statement differs from the REVOKE ENTRY statement in that it does not delete the entire entry from the ACL. The identifier remains, but without any privileges. An empty ACL entry denies all access to users matching the identifier, even if an entry later in the ACL grants PUBLIC access.

**ENTRY**
Deletes the entire entry in the ACL, including the identifier.

**ON DATABASE ALIAS alias**
**ON TABLE table-name**
**ON TABLE view-name**
**ON COLUMN column-name**
**ON MODULE module-name**
**ON FUNCTION ext-routine-name**
**ON PROCEDURE ext-routine-name**
Specifies whether the REVOKE statement applies to ACLs for databases, tables, views, columns, modules, or external routines. You can specify a list of names for any form of the ON clause. You must qualify a column name with at least the associated table name.

**FROM identifier**
**FROM PUBLIC**
Specifies the identifiers for the ACL entry to be modified or deleted. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

You can specify three types of identifiers:

- User identifiers
- General identifiers — (OpenVMS only)
- System-defined identifiers — (OpenVMS only)

OpenVMS VAX OpenVMS Alpha

You can specify more than one identifier by combining them with plus signs (+). Such identifiers are called multiple identifiers. They identify only those users who are common to all the groups defined by the individual identifiers. Users who do not match all the identifiers are not controlled by that entry.

For instance, the multiple identifier SECRETARIES + INTERACTIVE specifies only members of the group defined by the general identifier SECRETARIES that are interactive processes. It does not identify members of the SECRETARIES group that are not interactive processes. ♦

The following arguments briefly describe the three types of identifiers. For more information about identifiers, see your operating system documentation.

**user-identifier**

Uniquely identifies each user on the system.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the user identifier consists of the standard OpenVMS user identification code (UIC), a group name, and a member name (user name). The group name is optional. The user identifier can be in either numeric or alphanumeric format. The following are all valid user identifiers that could identify the same user:

```
K_JONES
[SYSTEM3, K_JONES]
[341,311] ♦
```

Digital UNIX

On Digital UNIX, a user identifier consists of the name of a group and the standard Digital UNIX user identifier (UID). The group name is optional. The user identifier must be in alphanumeric form. On Digital UNIX, users can belong to more than one group. The following are valid user identifiers that could identify the same user:

```
k_jones
[system3, k_jones] ♦
```

You can use the asterisk (*) wildcard character as part of a user identifier. For example, if you want to specify all users in a group on an OpenVMS system, you can enter [341,*] as the identifier.

When Oracle Rdb creates a database, it automatically creates an ACL entry with the identifier [*,*], which grants all privileges except DBCTRL to any user.

You cannot use more than one user identifier in a multiple identifier.

OpenVMS VAX OpenVMS Alpha

**general-identifier**
Identifies groups of users on the system and are defined by the OpenVMS system manager in the system privileges database. The following are possible general identifiers:

- DATAENTRY
- SECRETARIES
- MANAGERS ♦

OpenVMS VAX OpenVMS Alpha

**system-identifier**
Automatically defined by the OpenVMS system when the rights database is created at system installation time. System-defined identifiers are assigned depending on the type of login you execute. The following are all valid system-defined identifiers:

- BATCH
- NETWORK
- INTERACTIVE
- LOCAL
- DIALUP
- REMOTE ♦

**AFTER identifier**
**AFTER PUBLIC**
**POSITION n**
Specifies the position of the entry within the ACL. If you omit the AFTER or POSITION argument, SQL searches the entire ACL for an identifier list that matches the one specified in the FROM clause of the REVOKE statement. If it finds a match, it modifies the ACL entry by deleting the privileges specified in the privilege list. If there is no match, SQL generates an error and the REVOKE statement has no effect on the ACL.

Digital UNIX

On Digital UNIX, because users can belong to more that one group, SQL considers entries as matching only when the group and user name in both entries are identical. ♦

With the AFTER or POSITION argument, you can specify the position in the list from which SQL searches for an ACL entry with an identifier that matches the one specified in the FROM clause of the REVOKE statement.

- In the AFTER argument, the identifier specifies the entry in the ACL after which SQL begins its search for the entry to be modified or deleted.

If none of the entries in the ACL has an identifier that matches the identifier specified in the AFTER argument, SQL generates an error and the statement fails.

Starting after the entry specified by the identifier in the AFTER argument, SQL searches entries in the ACL. If an entry has an identifier that matches the identifier specified by the FROM clause of the REVOKE statement, SQL modifies or deletes that ACL entry.

If none of the entries has an identifier that matches the identifier specified by the FROM clause of the REVOKE statement, SQL generates an error and the statement fails (even if an entry before the position at which SQL began its search had an identifier that matched).

Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

- In the POSITION argument, the integer specifies the earliest relative position in the ACL of the entry to be modified or deleted. If the integer is larger than the number of entries in the ACL, SQL generates an error and the statement fails.

  Starting with the position specified by the POSITION argument, SQL searches entries in the ACL. If an entry has an identifier that matches the identifier specified by the FROM clause of the REVOKE statement, SQL modifies or deletes that ACL entry.

  If none of the entries has an identifier that matches the identifier specified by the FROM clause of the REVOKE statement, SQL generates an error and the statement fails (even if an entry before the position at which SQL began its search had an identifier that matched).

## Usage Notes

- Deletions and changes to ACLs do not take effect until you attach to the database again, even though those changes are displayed by the SHOW PROTECTION and SHOW PRIVILEGES statements.
- You must attach to all databases to which you refer in a REVOKE statement. If you use the default database attach, you must use the default alias (RDB$DBHANDLE in interactive and precompiled SQL; in SQL module language files, the identifier specified in the ALIAS clause) to work with database ACLs.

- Users with the DBADM database privilege can perform any data definition or data manipulation operation on any named object, including the database, regardless of the ACL for the object. The DBADM privilege is the most powerful privilege in Oracle Rdb because it can override most privilege checks performed by Oracle Rdb. Users with the DBADM database privilege implicitly receive *all* privileges for all objects, except the SECURITY database privilege.
- OpenVMS VAX OpenVMS Alpha Users with the OpenVMS SYSPRV privilege implicitly receive the same privileges as users with the DBADM database privilege ♦
- Digital UNIX On Digital UNIX, the users root and dbsmgr can perform any operation on any named object, including the database, regardless of the ACL of the object. Those users cannot be denied access to the database. They are implicitly granted all privileges.♦
- You cannot execute the REVOKE statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- You cannot REVOKE privileges on stored procedures or stored functions.

For more information on protection for an Oracle Rdb database, see the chapter on defining database privileges in the *Oracle Rdb7 Guide to Database Design and Definition*.

## Example

Example 1: Reattaching to make ACL changes take effect

```
SQL> -- Display the ACL for the EMPLOYEES table:
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
Protection on Table EMPLOYEES
    (IDENTIFIER=[sql,warring],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+
      DROP+DBADM+REFERENCES)
```

```
SQL>
SQL> -- User warring, the owner of the database, denies
SQL> -- herself INSERT access to the EMPLOYEES table:
SQL> REVOKE INSERT ON TABLE EMPLOYEES FROM warring;
SQL> COMMIT;
SQL> --
SQL> -- The SHOW PROTECTION statement displays the change
SQL> -- (INSERT is no longer part of the ACL entry for warring):
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
Protection on Table EMPLOYEES
    (IDENTIFIER=[sql,warring],ACCESS=SELECT+UPDATE+DELETE+SHOW+CREATE+ALTER+
      DROP+DBCTRL+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+ALTER+DROP)
SQL> --
SQL> -- But the change is not yet effective.
SQL> -- User warring can still store rows in the EMPLOYEES table:
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ("99999");
1 row inserted
SQL> SELECT EMPLOYEE_ID
cont>   FROM EMPLOYEES
cont>   WHERE EMPLOYEE_ID = "99999";
 EMPLOYEE_ID
 99999
1 row selected
SQL> ROLLBACK;
SQL>
SQL> -- To make the ACL change take effect, issue another ATTACH statement
SQL> -- to override the current attach:
SQL> ATTACH 'FILENAME personnel';
This alias has already been declared.
Would you like to override this declaration (No)? Y
SQL>
SQL> -- Now warring cannot insert new rows into the EMPLOYEES table:
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ("99999");
%RDB-E-NO_PRIV, privilege denied by database facility
SQL>
SQL> -- A GRANT statement gives all privileges back to WARRING:
SQL> GRANT ALL ON TABLE EMPLOYEES TO warring;
SQL> COMMIT;
```

## REVOKE Statement, ANSI/ISO-Style

Removes privileges from the Oracle Rdb access privilege set granted by a specific user for a database, table, column, module, or external routine. Each entry in an ANSI/ISO-style access privilege set consists of an identifier and a list of privileges assigned to the identifier.

- Each identifier specifies a user or the PUBLIC keyword.
- The set of privileges specifies what operations that user or user group can perform on the database, table, column, module, or external routine.

For ANSI/ISO-style databases, the access privilege set is not order-dependent. The user matches the entry in the access privilege set, receives whatever privileges have been granted on the database, table, column, module, or external routine, and receives the privileges defined for PUBLIC. A user without an entry in the access privilege set receives only the privileges defined for PUBLIC. A user loses a privilege when there are no users who grant that privilege to the user. The PUBLIC identifier always has an entry in the access privilege set, even if PUBLIC has no access to the database, table, column, module, or external routine.

To create or add privileges to an entry to the Oracle Rdb access privilege set for a database, table, view, column, module, or external routine, see the GRANT Statement, ANSI/ISO-Style.

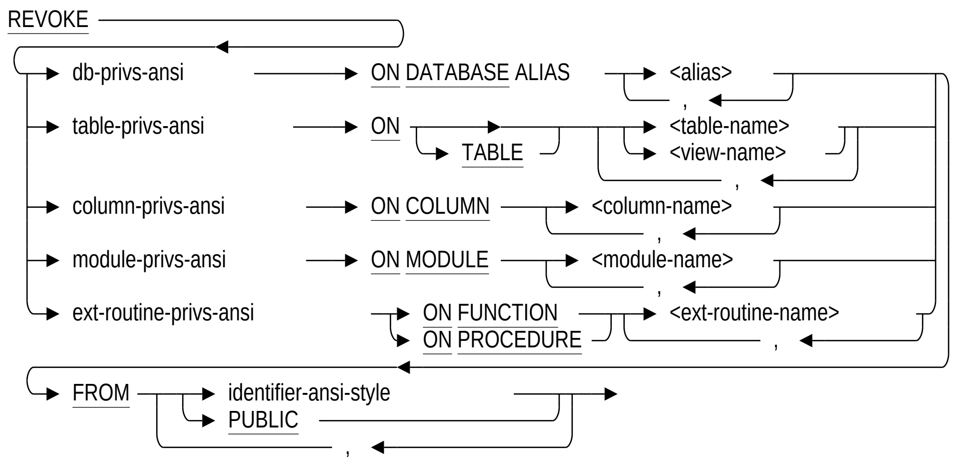
### Environment

You can use the REVOKE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a nonstored procedure in a nonstored SQL module
- In dynamic SQL as a statement to be dynamically executed

## REVOKE Statement, ANSI/ISO-Style

### Format



db-privs-ansi =

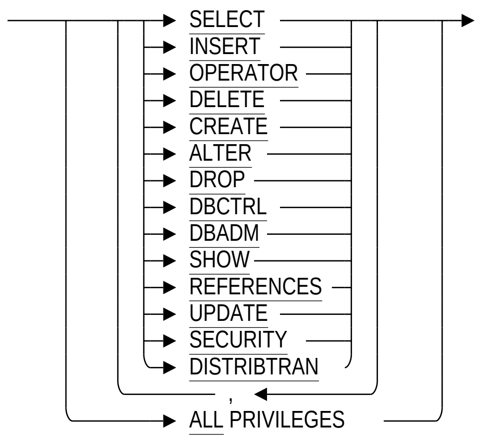
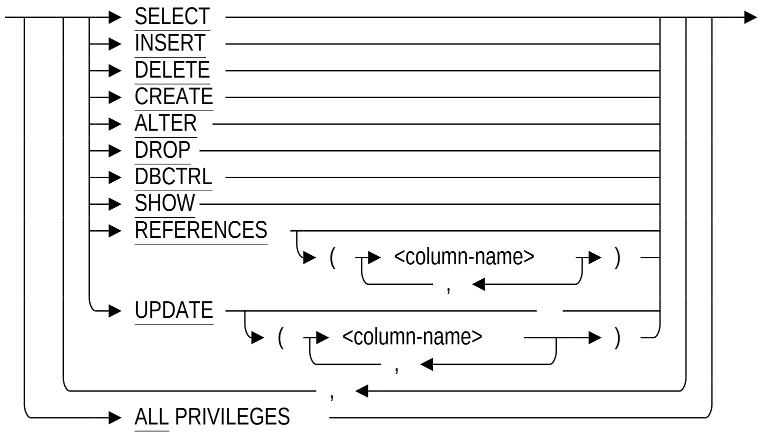
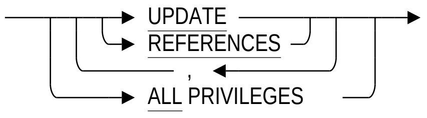


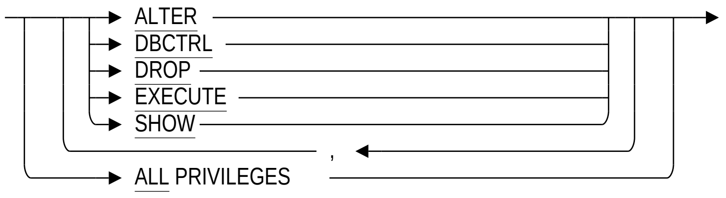
table-privs-ansi =



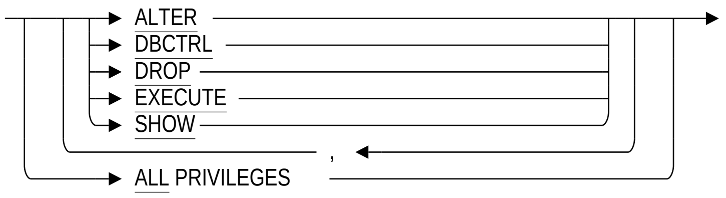
column-privs-ansi =

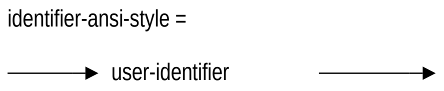


module-privs-ansi =



ext-routine-privs-ansi =





## Arguments

**db-privs-ansi**
**table-privs-ansi**
**column-privs-ansi**
**module-privs-ansi**
**ext-routine-privs-ansi**
Specifies the list of privileges you want to remove from an existing access privilege set entry. The operations permitted by a given privilege keyword differ, depending on whether it was granted for a database, table, column, module, or external routine. Table 7–1 in the GRANT Statement lists the privilege keywords and their meanings for databases, tables, modules, and external routines.

**ALL PRIVILEGES**
Specifies that SQL should revoke all privileges in the access privilege set entry.

**ON DATABASE ALIAS alias**
**ON TABLE table-name**
**ON TABLE view-name**
**ON COLUMN column-name**
**ON MODULE module-name**
**ON FUNCTION ext-routine-name**
**ON PROCEDURE ext-routine-name**
Specifies whether the REVOKE statement applies to access privilege sets for databases, tables, views, columns, modules, or external routines. You can specify a list of names for any form of the ON clause. You must qualify a column name with at least the associated table name.

**FROM identifier-ansi-style**
**FROM PUBLIC**
Specifies the identifiers for the access privilege set entry to be modified or deleted. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

The only identifiers are ones that translate to an OpenVMS user identification code (UIC) or a Digital UNIX user identifier (UID).

For more information about user identifiers, see the operating system documentation.

**user-identifier**
Uniquely identifies each user on the system.

OpenVMS VAX OpenVMS Alpha On OpenVMS, the user identifier consists of the standard OpenVMS user identification code (UIC), a group name, and a member name (user name). The group name is optional. The user identifier can be in either numeric or alphanumeric format. The following are all valid user identifiers that could identify the same user:

K_JONES
[SYSTEM3, K_JONES]
[341,311] ♦

Digital UNIX On Digital UNIX, a user identifier consists of the name of a group and the standard Digital UNIX user identifier (UID). The group name is optional. The user identifier must be in alphanumeric form. On Digital UNIX, users can belong to more than one group. The following are all valid user identifiers that could identify the same user:

k_jones
[system3, k_jones] ♦

When Oracle Rdb creates a database, it automatically creates an access privilege set entry with the PUBLIC identifier, which grants all privileges except DBCTRL to any user. In access privilege set databases, the only wildcard allowed is the PUBLIC identifier.

You cannot use more than one user identifier in a multiple identifier.

## Usage Notes

- Deletions and changes to access privilege sets do not take effect until you attach to the database again, even though those changes are displayed by the SHOW PROTECTION and SHOW PRIVILEGES statements.
- You must attach to all databases to which you refer in a REVOKE (ANSI/ISO-style) statement. If you use the default database attach, you must use the default alias (RDB$DBHANDLE in interactive and precompiled SQL; in SQL module language files, the identifier specified in the ALIAS clause) to work with database access privilege sets.

OpenVMS VAX OpenVMS Alpha

- The OpenVMS privileges SYSPRV and BYPASS override privileges in access privilege set entries. Users with either of those OpenVMS privileges cannot be denied access to the database. They are implicitly granted all privileges.♦

Digital UNIX

- On Digital UNIX, the users root and dbsmgr can perform any operation on any named object, including the database, regardless of the ACL of the object. Those users cannot be denied access to the database. They are implicitly granted all privileges. ♦
- You can revoke only column-level privileges that have been specifically granted at the column level.
- For the SELECT, INSERT, and DELETE data manipulation privileges, SQL checks the access privilege set for the database and for the individual table before allowing access to a specific table. For example, if your SELECT privilege for a database that contains the EMPLOYEES table is revoked, you will not be able to read rows from the table even though you may have SELECT privilege to the EMPLOYEES table itself.
- To revoke the data manipulation privileges UPDATE and REFERENCES, you need to have been granted at least read access to the database and the appropriate column privilege.
- When a privilege is revoked from the grantee who received the privilege with the WITH GRANT OPTION clause, the privilege is also revoked from all users who received the privilege from that grantee (unless these users have received the privilege from yet another user who still has the privilege).
- You cannot execute the REVOKE statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the RDB$SYSTEM storage area.
- You cannot REVOKE privileges on stored procedures or stored functions.

For more information on protection for an Oracle Rdb database, see the chapter on defining database privileges in the *Oracle Rdb7 Guide to Database Design and Definition*.

## Examples

Example 1: Reattaching to make access privilege set changes take effect

```
SQL> -- Display the access privilege set for the EMPLOYEES table:
SQL> SHOW PROTECTION ON EMPLOYEES;
Protection on Table EMPLOYEES
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
[sql,warring]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,
                            DROP,DBCTRL,DBADM,REFERENCES
SQL> -- User warring, the owner of the database, denies
SQL> -- herself INSERT access to the EMPLOYEES table.
SQL> REVOKE INSERT ON EMPLOYEES FROM warring;
SQL> COMMIT;
SQL> -- The SHOW PROTECTION statement displays the change
SQL> -- (INSERT is no longer part of the access privilege set entry
SQL> -- for warring):
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
[sql,warring]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,DBADM,REFERENCES
SQL>
SQL> -- But the change is not yet effective.
SQL> -- User warring can still store rows in the EMPLOYEES table.
SQL> -- To make the access privilege set change take effect,
SQL> -- issue another ATTACH statement to override the current attach.
SQL> ATTACH 'FILENAME ansi_test';
This alias has already been declared.
Would you like to override this declaration (No)? Y
SQL> -- Now warring cannot insert new rows into the
SQL> -- EMPLOYEES table.
SQL> INSERT INTO EMPLOYEES (EMPLOYEE_ID) VALUES ("99999");
%RDB-E-NO_PRIV, privilege denied by database facility
SQL>
SQL> -- A GRANT statement gives all privileges back to warring.
SQL> GRANT ALL ON TABLE EMPLOYEES TO warring;
SQL> COMMIT;
```

Example 2: Revoking a privilege granted with the WITH GRANT OPTION clause

When the privilege is revoked from the grantee, rdb_doc, who received the privilege with the WITH GRANT OPTION clause, the privilege is also revoked from all users who received the privilege from that grantee.

```
SQL> SHOW PROTECTION ON TABLE EMPLOYEES;
[*,*]:
  With Grant Option:       NONE
  Without Grant Option:    SELECT
[sql,warring]:
  With Grant Option:       SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,
                           DROP,DBCTRL,OPERATOR,DBADM,REFERENCES
  Without Grant Option:    SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,
                           DROP,DBCTRL,DBADM,REFERENCES
[rdb,rdb_doc]:
  With Grant Option:       SHOW
  Without Grant Option:    NONE
SQL>
SQL> REVOKE SHOW ON EMPLOYEES FROM [rdb,rdb_doc];
SQL> SHOW PROTECTION ON EMPLOYEES;
Protection on Table EMPLOYEES
[*,*]:
  With Grant Option:       NONE
  Without Grant Option:    SELECT
[rdb,rdb_doc]:
  With Grant Option:       NONE
  Without Grant Option:    NONE
```

Example 3: Revoking column privileges

This example shows how to restrict privileges on a specific column by revoking the UPDATE privilege that has been granted for that column.

```
SQL> SHOW PROTECTION ON COLUMN EMPLOYEES.EMPLOYEE_ID;
[rdb,rdb_doc]:
  With Grant Option:       NONE
  Without Grant Option:    UPDATE
SQL> REVOKE UPDATE ON COLUMN EMPLOYEES.EMPLOYEE_ID FROM [rdb,rdb_doc];
SQL> SHOW PROTECTION ON COLUMN EMPLOYEES.EMPLOYEE_ID;
[rdb,rdb_doc]:
  With Grant Option:       NONE
  Without Grant Option:    NONE
```

## ROLLBACK Statement

Ends a transaction and undoes all changes you made since that transaction began. The ROLLBACK statement also:

- Closes all open cursors
- Releases all locks
- Performs a checkpoint operation if fast commit processing is enabled

The ROLLBACK statement affects:

- All open databases included in the current transaction
- All changes to data made with SQL data manipulation statements (DELETE, UPDATE, and INSERT)
- All changes to data definitions made with SQL data definition statements (ALTER, CREATE, and DROP)

### Environment

You can use the ROLLBACK statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
ROLLBACK WORK
```

### Arguments

**WORK**
Specifies an optional keyword that has no effect on the ROLLBACK statement. It is provided for compatibility with the ANSI/ISO SQL standard.

## Usage Notes

You cannot use the ROLLBACK statement in an ATOMIC compound statement.

## Example

Example 1: Rolling back changes in a COBOL program

```
GET-ID-NUMBER.
      DISPLAY "Enter employee ID number:  "
         WITH NO ADVANCING.
      ACCEPT EMPLOYEE-ID.
CHANGE-SALARY.
      DISPLAY "Enter new salary amount:  "
         WITH NO ADVANCING.
      ACCEPT SALARY-AMOUNT.

EXEC SQL  UPDATE  SALARY_HISTORY
          SET     SALARY_AMOUNT = :SALARY-AMOUNT
          WHERE   EMPLOYEE_ID = :EMPLOYEE-ID
          AND     END_DATE IS NULL
END-EXEC

      DISPLAY EMPLOYEE-ID, SALARY-AMOUNT.
      DISPLAY "Is this figure correct? [Y or N]  "
         WITH NO ADVANCING.
      ACCEPT ANSWER.
      IF ANSWER = "Y" THEN

        EXEC SQL  COMMIT END-EXEC
      ELSE
        EXEC SQL  ROLLBACK END-EXEC
        DISPLAY "Please enter the new salary amount again."
        GO TO CHANGE-SALARY
      END-IF.
```

## SELECT Statement: General Form

Specifies a result table. A **result table** is an intermediate table of values derived from columns and rows of one or more tables or views that meet conditions specified by a select expression. The tables or views that the columns and rows come from are identified in the FROM clause of the statement.

The basic element of a SELECT statement is called a select expression. Section 2.8.1 describes select expressions in detail.

You can use the general form of the SELECT statement only in interactive and dynamic SQL. To retrieve rows of a result table in host language programs, you must use the DECLARE CURSOR statement or a special form of SELECT statement called a singleton select. See the SELECT Statement: Singleton Select for more information about a singleton select.

SQL evaluates the clauses of a SELECT statement in the following order:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. Select list
6. ORDER BY
7. LIMIT TO
8. OPTIMIZE

After each of these clauses, SQL produces an intermediate result table that is used in evaluating the next clause.

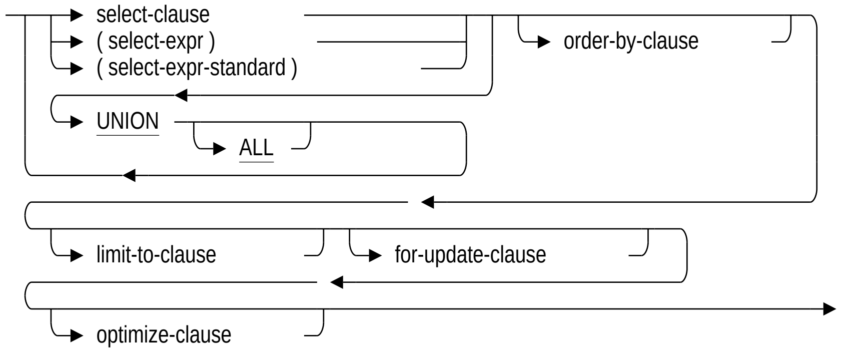
### Environment

You can use the general form of the SELECT statement only in interactive and dynamic SQL.

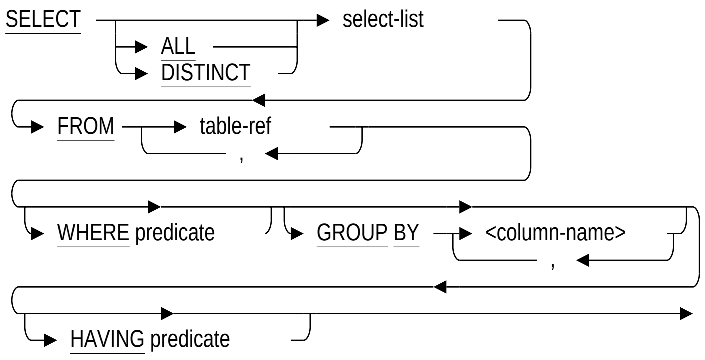
## SELECT Statement: General Form

## Format

select-expr =



select-clause =



select-list =

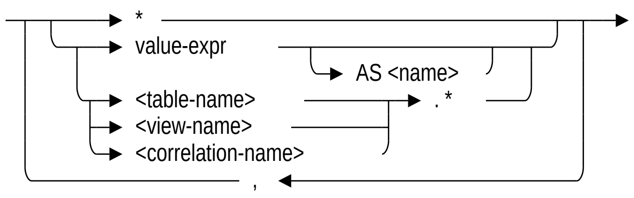
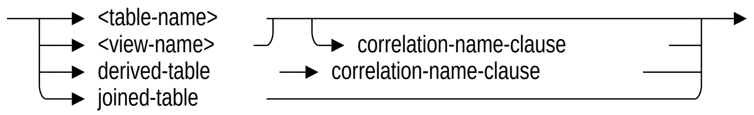
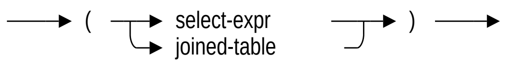


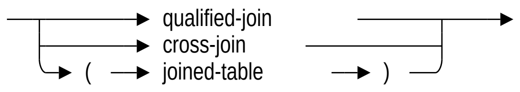
table-ref =



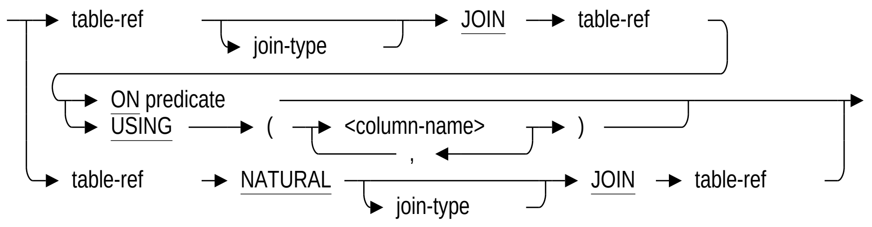
derived-table =



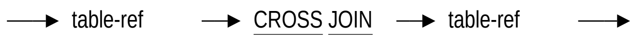
joined-table =



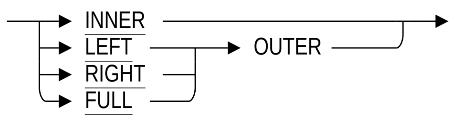
qualified-join =



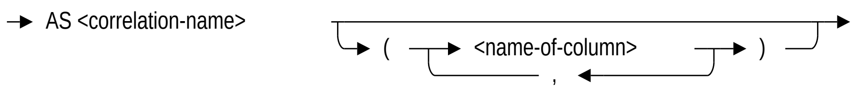
cross-join =



join-type =

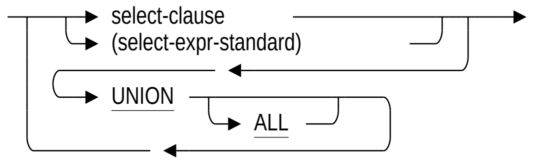


correlation-name-clause =

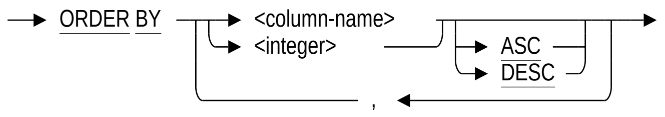


## SELECT Statement: General Form

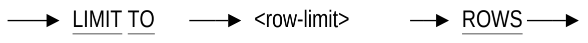
select-expr-standard =



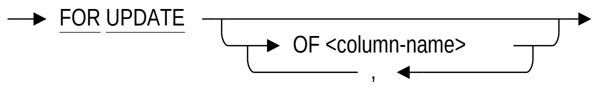
order-by-clause =



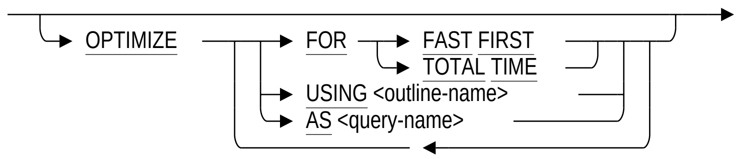
limit-to-clause =



for-update-clause =



optimize-clause =



## Arguments

### select-expr

See Section 2.8.1 for a detailed description of select expressions.

### FOR UPDATE OF column-name

Specifies the columns in a cursor that you or your program might later modify with an UPDATE statement. The column names in the FOR UPDATE clause must belong to a table or view named in the FROM clause.

You do not have to specify the FOR UPDATE clause of the SELECT statement to later modify rows using the UPDATE statement:

- If you do specify a FOR UPDATE clause with column names and later specify columns in the UPDATE statement that are not in the FOR UPDATE clause, SQL issues a warning message and proceeds with the update modifications.
- If you do specify a FOR UPDATE clause but do not specify any column names, you can update any column using the UPDATE statement. SQL does not issue any messages.
- If you do not specify a FOR UPDATE clause, you can update any column using the UPDATE statement. SQL does not issue any messages.

If you have set your dialect to ORACLE LEVEL1, the FOR UPDATE OF clause in a SELECT statement provides UPDATE ONLY CURSOR semantics by locking all the rows selected; which is a different semantic of the same feature in the Oracle7 server.

**OPTIMIZE FOR**
Specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- FAST FIRST

  A query optimized for FAST FIRST returns data to the user as quickly as possible, even at the expense of total throughput.

  If a query can be cancelled prematurely, you should specify FAST FIRST optimization. A good candidate for FAST FIRST optimization is an interactive application that displays groups of records to the user, where the user has the option of aborting the query after the first few screens. For example, singleton SELECT statements default to FAST FIRST optimization.

  If optimization strategy is not explicitly set, FAST FIRST is the default.
- TOTAL TIME

  If your application runs in batch, accesses all the records in the query, and performs updates or writes a report, you should specify TOTAL TIME optimization. Most queries benefit from TOTAL TIME optimization.

  The following examples illustrate the DECLARE CURSOR syntax for setting a preferred optimization mode:

```
SQL> DECLARE TEMP1 TABLE CURSOR
cont>  FOR
cont>    SELECT *
cont>      FROM EMPLOYEES
cont>      WHERE EMPLOYEE_ID > '00400'
cont>  OPTIMIZE FOR FAST FIRST;
SQL> --
SQL> DECLARE TEMP2 TABLE CURSOR
cont>  FOR
cont>    SELECT LAST_NAME, FIRST_NAME
cont>      FROM EMPLOYEES
cont>        ORDER BY LAST_NAME
cont>  OPTIMIZE FOR TOTAL TIME;
```

**OPTIMIZE USING outline-name**

Explicitly names the query outline to be used with the select expression even if the outline ID for the select expression and for the outline are different.

The following example is the query used to create an outline named WOMENS_DEGREES:

```
SQL> SELECT E.LAST_NAME, E.EMPLOYEE_ID, D.DEGREE, D.DEGREE_FIELD, D.YEAR_GIVEN
cont> FROM EMPLOYEES E, DEGREES D WHERE E.SEX = 'F'
cont> AND E.EMPLOYEE_ID = D.EMPLOYEE_ID
cont> ORDER BY LAST_NAME
```

By using the OPTIMIZE USING clause and specifying the WOMENS_DEGREES outline, you can ensure that Oracle Rdb attempts to use the WOMENS_DEGREES outline to execute a query even if the query is slightly different as shown in the following example:

```
SQL> SELECT E.LAST_NAME, E.EMPLOYEE_ID, D.DEGREE, D.DEGREE_FIELD, D.YEAR_GIVEN
cont> FROM EMPLOYEES E, DEGREES D WHERE E.SEX = 'F'
cont> AND E.EMPLOYEE_ID = D.EMPLOYEE_ID
cont> ORDER BY LAST_NAME
cont> LIMIT TO 10 ROWS
cont> OPTIMIZE USING WOMENS_DEGREES;
~S: Outline WOMENS_DEGREES used  <-- the query uses the WOMENS_DEGREES outline
   .
   .
   .
```

```
 E.LAST_NAME      E.EMPLOYEE_ID   D.DEGREE   D.DEGREE_FIELD    D.YEAR_GIVEN
 Boyd             00244           MA         Elect. Engrg.             1982
 Boyd             00244           PhD        Applied Math              1979
 Brown            00287           BA         Arts                      1982
 Brown            00287           MA         Applied Math              1979
 Clarke           00188           BA         Arts                      1983
 Clarke           00188           MA         Applied Math              1976
 Clarke           00196           BA         Arts                      1978
 Clinton          00235           MA         Applied Math              1975
 Clinton          00201           BA         Arts                      1973
 Clinton          00201           MA         Applied Math              1978
10 rows selected
```

See the CREATE OUTLINE Statement for more information on creating an outline.

**OPTIMIZE AS query-name**

Assigns a name to the query. You must define the RDMS$DEBUG_FLAGS logical name or the RDB_DEBUG_FLAGS configuration parameter be the letter *S* to see the access methods used to produce the results of the query. The following example shows how to use the OPTIMIZE AS clause:

```
SQL> DELETE FROM EMPLOYEES E
cont> WHERE EXISTS ( SELECT *
cont>                FROM   SALARY_HISTORY S
cont>                WHERE  S.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>                AND    S.SALARY_AMOUNT > 75000)
cont> OPTIMIZE AS DEL_EMPLOYEE;
Leaf#01 FFirst RDB$RELATIONS Card=19
   .
   .
   .
~Query Name : DEL_EMPLOYEE
   .
   .
   .
7 rows deleted
```

## Usage Notes

- The FOR UPDATE OF clause provides UPDATE ONLY CURSOR semantics by locking all the rows selected if you have set your dialect to ORACLE LEVEL1.
- If an outline exists, Oracle Rdb uses the outline specified in the OPTIMIZE USING clause unless one or more of the directives in the outline cannot be followed. For example, if the compliance level for the outline is mandatory and one of the indexes specified in the outline directives has been deleted, the outline is not used. SQL issues an error message if an existing outline cannot be used.

  If you specify the name of an outline that does not exist, Oracle Rdb compiles the query, ignores the outline name, and searches for an existing outline with the same outline ID as the query. If an outline with the same outline ID is found, Oracle Rdb attempts to execute the query using the directives in that outline. If an outline with the same outline ID is not found, the optimizer selects a strategy for the query for execution.

  See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information regarding query outlines.

## Examples

Example 1: Using the SELECT statement

The following SELECT statement returns all rows from the EMPLOYEES table in no specific order:

```
SQL> SELECT LAST_NAME, FIRST_NAME, MIDDLE_INITIAL FROM EMPLOYEES;
 LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 Toliver          Alvin        A
 Smith            Terry        D
 Dietrich         Rick         NULL
 Kilpatrick       Janet        NULL
    .
    .
    .
100 rows selected
```

Example 2: Adding an ORDER BY clause to sort rows selected

An ORDER BY clause added to the same SELECT statement causes SQL to sort the rows according to the LAST_NAME column.

```
SQL> SELECT LAST_NAME, FIRST_NAME, MIDDLE_INITIAL FROM
cont> EMPLOYEES ORDER BY LAST_NAME;
 LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 Ames             Louie        A
 Andriola         Leslie       Q
 Babbin           Joseph       Y
 Bartlett         Dean         G
 Bartlett         Wes          NULL
   .
   .
   .
100 rows selected
```

Example 3: Adding a LIMIT TO clause to return a certain number of rows

The same SELECT statement with both an ORDER BY clause and a LIMIT TO clause causes SQL to:

1. Sort all the rows of the EMPLOYEES table according to the LAST_NAME column
2. Return the first five rows in the ordered set

```
SQL> SELECT LAST_NAME, FIRST_NAME, MIDDLE_INITIAL FROM
cont> EMPLOYEES ORDER BY LAST_NAME LIMIT TO 5 ROWS;
 LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
 Ames             Louie        A
 Andriola         Leslie       Q
 Babbin           Joseph       Y
 Bartlett         Dean         G
 Bartlett         Wes          NULL
5 rows selected
```

Example 4: Using the optimize clause to specify an outline and a query name

The following select query uses a previously defined outline called WOMENS_DEGREES and also names the query. The RDMS$DEBUG_FLAGS logical has been set to “Ss”:

```
SQL> SELECT E.LAST_NAME, E.EMPLOYEE_ID, D.DEGREE,
cont> D.DEGREE_FIELD, D.YEAR_GIVEN
cont>    FROM EMPLOYEES E, DEGREES D
cont>    WHERE E.SEX = 'F'
cont>       AND E.EMPLOYEE_ID = D.EMPLOYEE_ID
cont>    ORDER BY LAST_NAME
cont>    OPTIMIZE USING WOMENS_DEGREES
cont>             AS WOMENS_DEGREES;
~Query Name : WOMENS_DEGREES
~S: Outline WOMENS_DEGREES used
Sort
Cross block of 2 entries
  Cross block entry 1
    Conjunct       Get     Retrieval by index of relation EMPLOYEES
      Index name  EMP_EMPLOYEE_ID [0:0]
  Cross block entry 2
    Leaf#01 BgrOnly DEGREES Card=165
      BgrNdx1 DEG_EMP_ID [1:1] Fan=17
-- Rdb Generated Outline : 16-JUN-1994 11:01
create outline WOMENS_DEGREES
id 'D3A5BC351F507FED820EB704FC3F61E8'
mode 0
as (
  query (
    subquery (
      EMPLOYEES 0     access path index       EMP_EMPLOYEE_ID
        join by cross to
      DEGREES 1       access path index       DEG_EMP_ID
      )
    )
  )
compliance optional     ;
 E.LAST_NAME      E.EMPLOYEE_ID   D.DEGREE   D.DEGREE_FIELD    D.YEAR_GIVEN
 Boyd             00244           MA         Elect. Engrg.             1982
 Boyd             00244           PhD        Applied Math              1979
 Brown            00287           BA         Arts                      1982
 Brown            00287           MA         Applied Math              1979
 Clarke           00188           BA         Arts                      1983
 Clarke           00188           MA         Applied Math              1976
 Clarke           00196           BA         Arts                      1978
   .
   .
   .
61 rows selected
```

## SELECT Statement: Singleton Select

Specifies a result table. A **result table** is an intermediate table of values derived from columns and rows of one or more tables or views that meet conditions specified by a select expression. The tables or views that the columns and rows come from are identified in the FROM clause of the statement.

The basic element of a SELECT statement is called a select expression. Section 2.8.1 describes select expressions in detail.

To retrieve rows of a result table in host language programs, you must use the DECLARE CURSOR statement or a special form of SELECT statement called a singleton select. A **singleton select** statement specifies a one-row result table, and is allowed in either precompiled programs or as part of a procedure in an SQL module. A singleton select includes an additional clause, INTO, to assign the values in the row to host language variables in a program. A singleton select, also called an embedded select, is not allowed in interactive SQL.

For information on the general form of the SELECT statement, see the SELECT Statement: General Form.

### Environment

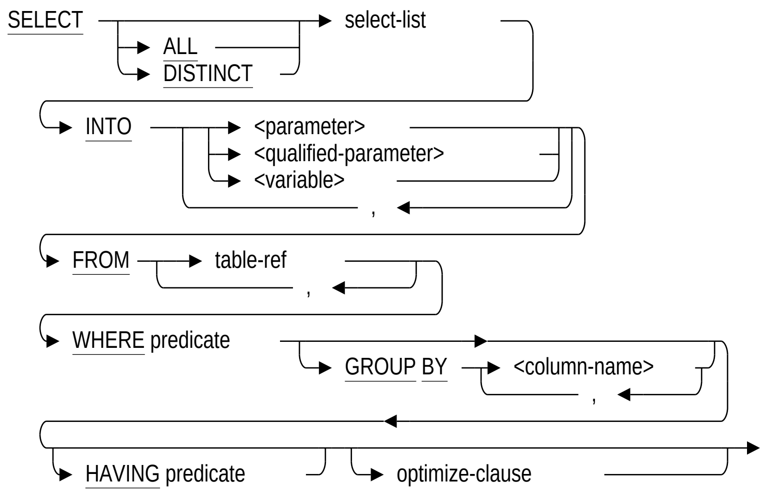
You can use a singleton select statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQl as a statement to be dynamically executed

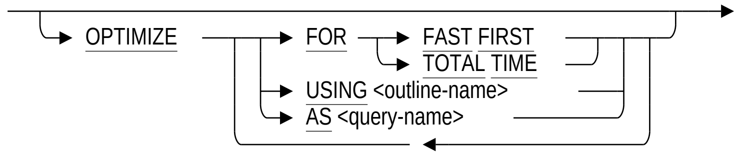
## SELECT Statement: Singleton Select

### Format

singleton-select =



optimize-clause =



### Arguments

**ALL**
Specifies that duplicate rows should not be eliminated from the result table. ALL is the default.

**DISTINCT**
Specifies that SQL should eliminate duplicate rows from the result table.

**select-list**
For a description of select lists, see Section 2.8.1.

**INTO parameter**
**INTO qualified-parameter**
**INTO variable**
Specifies a list of parameters, qualified parameters (structures), or variables to receive values from the columns of the one-row result table. The variables named must have been declared in the host program. If a variable named in the list is a host structure, SQL considers the reference the same as a reference to each of the elements of the host structure.

If the number of variables specified, either explicitly or by reference to a host structure, does not match the number of values in the row of the result table, SQL generates an error when it precompiles the program or compiles the SQL module file.

If columns in the result table from a singleton select include null values, the corresponding parameters must include indicator parameters.

**FROM table-ref**
For a description of the FROM clause, see Section 2.8.1.

**WHERE predicate**
For a description of the WHERE clause, see Section 2.8.1.

**GROUP BY column-name**
For a description of the GROUP BY clause, see Section 2.8.1.

**HAVING predicate**
For a description of the HAVING clause, see Section 2.8.1.

**optimize-clause**
For a description of the OPTIMIZE clause, see Section 2.8.1.

## Usage Notes

- The following restrictions distinguish a singleton select from a SELECT statement. A singleton select cannot:
  - Specify a result table that is longer than a single row (SQL generates an error if it does)
  - Omit the INTO clause
- To ensure that only one row is returned with a SINGLETON SELECT statement, use the LIMIT TO 1 ROW clause. For more information on the LIMIT TO clause, see Section 2.8.1.

## SET Statement

Changes the characteristics of SQL terminal sessions. You can control the:

- Currency indicator to be displayed for output

OpenVMS VAX OpenVMS Alpha

- Display format for date values, time values, or both
- Default path name in the data dictionary ♦
- Digit separator to be displayed for output

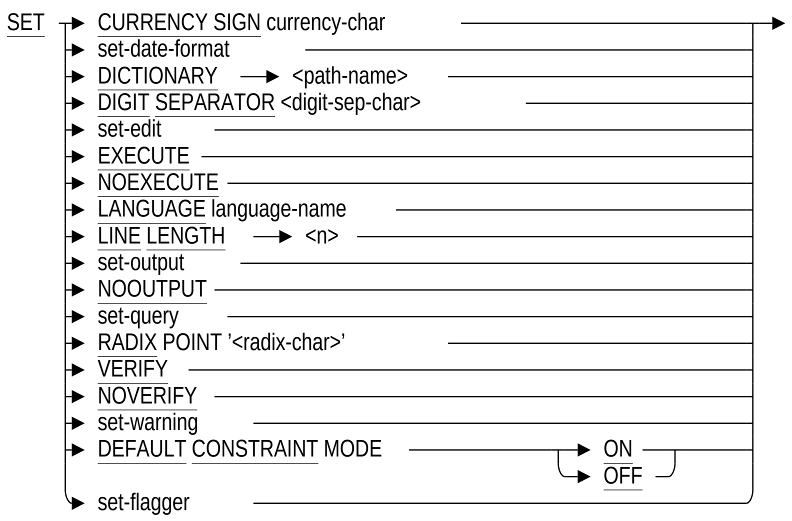
OpenVMS VAX OpenVMS Alpha

- Number of statements to be included in the editing buffer when you type EDIT *
- Language to be used for month abbreviations, and so on, in date and time input and display ♦
- Length of lines to be displayed for output
- File in which the session is recorded
- Preview of query costs before any rows are actually returned
- Number of rows output, the number of seconds allowed per query compilation and execution, or the amount of CPU time expended for each query compilation and execution
- Character used to display the radix point in output
- Display of statements from a command file
- Display of warning messages about deprecated features
- Default constraint setting for statements
- Display of warning messages about nonstandard syntax

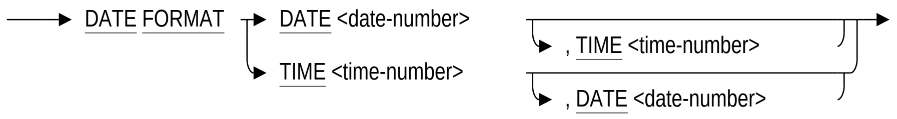
### Environment

You can use the SET statement in interactive SQL only.

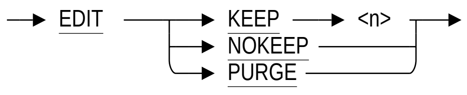
## Format



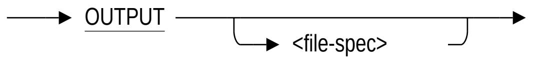
set-date-format=



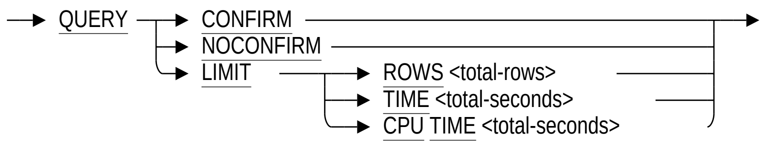
set-edit=



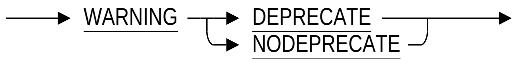
set-output=



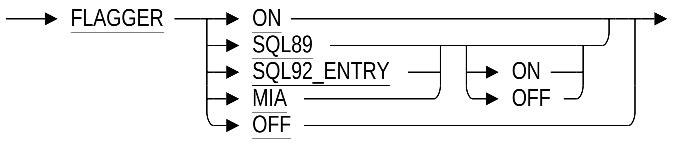
set-query =



set-warning=



set-flagger =



## Arguments

**CURRENCY SIGN currency-char**
Specifies the currency indicator to be displayed in output. (SQL produces currency indicators in output when you specify the dollar sign ($) edit string for the column. See Section 2.5.2 for more information on edit strings.)

If you do not specify an alternate character, the default is either the dollar sign ($) or, on OpenVMS, the value specified by the logical name SYS$CURRENCY.

Digital UNIX

When you use Oracle Rdb for Digital UNIX, you must specify a currency character. ♦

OpenVMS VAX OpenVMS Alpha

**DATE FORMAT**
Specifies the display format for either date values, time values, or both.

You must specify a numeric argument with the DATE and TIME portions of the SET DATE FORMAT statement. This numeric argument is the same as the numeric portion of certain OpenVMS Run-Time Library formats. The formats are documented in the OpenVMS run-time library documentation. (This statement only accepts numbers that reference OpenVMS format date and time logical names; it does not support the ANSI/ISO date and time data types.)

The SET DATE FORMAT DATE and SET DATE FORMAT TIME statements change only the output for the date or time formats. If you want to change the input format, use the logical name LIB$DT_INPUT_FORMAT. You must run the command procedure SYS$MANAGER:LIB$DT_STARTUP.COM before using any of the run-time library date-time routines for input or output formats other than the default. The LIB$DT_STARTUP.COM procedure also defines spellings for date and time elements in languages other than English. See the

OpenVMS run-time library documentation for more information on LIB$DT_INPUT_FORMAT.

The SET DATE FORMAT statement is available only on OpenVMS platforms. ♦

OpenVMS VAX OpenVMS Alpha

**DATE date-number**

Specifies the display format for date values.

You must enter a number for the date-number argument. This number corresponds to numbers in the date format logical names listed in tables in the OpenVMS run-time library documentation.

For example, LIB$DATE_FORMAT_006 is one of the logical names in the table. The logical name specifies the format in which the eighth day of May in the year 1957 would be displayed as 8 May 57. Note that the latter part of the logical name is the number 006.

If you wanted to specify the 8 May 57 format using the SET DATE FORMAT statement, you would use the numeric part of the LIB$DATE_FORMAT_006 logical name, 6. You do not have to enter any leading zeros that the number might have.

If you do not specify a date format, the default is dd-mmm-yyyy.

The SET DATE FORMAT DATE statement is available only on OpenVMS platforms. ♦

OpenVMS VAX OpenVMS Alpha

**TIME time-number**

Specifies the display format for time values.

You must enter a number for the time-number argument. This number corresponds to numbers in the time-format logical names listed in tables in the OpenVMS run-time library documentation.

For example, the table contains the logical name LIB$TIME_FORMAT_020. The logical name specifies the format in which the eighth hour, fourth minute, and thirty-second second of a day would be displayed as 8 h 4 min 32 s. Note that the latter part of the logical name is the number 020.

If you wanted to specify the 8 h 4 min 32 s format for the SQL SET DATE FORMAT TIME statement, you would use the numeric part of the LIB$TIME_FORMAT_020 logical name, 20. You do not have to enter any leading zeros that the number might have.

If you do not specify a time format, the default is hh:mm:ss.cc.

The SET DATE FORMAT TIME statement is available only on OpenVMS platforms. ♦

OpenVMS VAX OpenVMS Alpha

**DICTIONARY path-name**
Changes your default repository path name to the path name you specify.

The SET DICTIONARY statement is available only on OpenVMS platforms. ♦

**DIGIT SEPARATOR digit-sep-char**
Changes the output displaying the digit separator to the specified character. The digit separator is the symbol that separates groups of three digits in values greater than 999. For example, the comma is the digit separator in the number 1,000.

(SQL produces digit separators in output when you specify the comma (,) edit string for the column. See Section 2.5.2 for more information on edit strings.)

You must enclose the digit-sep-char argument within single quotation marks.

If you do not specify an alternate character, the default is either the comma (,) or, on OpenVMS, the value specified by the logical name SYS$DIGIT_SEP.

Digital UNIX

When you use Oracle Rdb for Digital UNIX, you must specify a digit separator character. ♦

OpenVMS VAX OpenVMS Alpha

**EDIT**
Controls the size of the editing buffer that you create when you use the EDIT statement with a wildcard as the argument.

- SET EDIT KEEP n

  Tells SQL to save the previous *n* statements. For example, assume you have specified SET EDIT KEEP 5. When you type EDIT *, SQL places the previous five statements in the editing buffer. The number you specify with SET EDIT KEEP is the maximum number of statements you can recall with the EDIT statement. The default is 20.

- SET EDIT NOKEEP

  This statement is equivalent to SET EDIT KEEP 0. If you use this form of the statement and you type EDIT or EDIT *, your editing buffer will be empty. This form of the statement saves system resources when you are running command files rather than an interactive process.

- SET EDIT PURGE

  This statement retains the value of the KEEP parameter but purges all previous statements. As with SET EDIT NOKEEP, if you use the SET EDIT PURGE statement and then EDIT or EDIT *, your editing buffer will be empty. Unlike the SET EDIT NOKEEP statement, however, SET EDIT PURGE causes SQL to accumulate subsequent statements to place in the editing buffer when you issue EDIT statements later in the interactive session.

The SET EDIT statement is available only on OpenVMS platforms. ♦

**EXECUTE**
**NOEXECUTE**
Instructs SQL whether to execute the data manipulation statements you issue in an interactive SQL session. See the Examples to see how you could use the NOEXECUTE option to check for proper syntax before you issue a statement against a database.

You can use the NOEXECUTE option in conjunction with RDMS$DEBUG_FLAGS to examine the estimated cost and access strategy associated with a query. If you specify SET NOEXECUTE, SQL displays the access strategies without executing the query. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for information on using RDMS$DEBUG_FLAGS.

If you do not specify EXECUTE or NOEXECUTE, the default is EXECUTE.

OpenVMS VAX OpenVMS Alpha

**LANGUAGE language-name**
Specifies the language to be used for translation of month names and abbreviations in date and time input and display. The language-name argument also determines the translation of other language-dependent text, such as the translation for the date literals YESTERDAY, TODAY, and TOMORROW.

If you do not specify a language, the default is the language specified by the logical name SYS$LANGUAGE. If you require different language spellings, you must define the logical name SYS$LANGUAGES in addition to SYS$LANGUAGE. You must run the command procedure SYS$MANAGER:LIB$DT_STARTUP.COM after defining SYS$LANGUAGES. For example:

```
$ DEFINE SYS$LANGUAGES FRENCH, GERMAN, SPANISH
$ RUN SYS$MANAGER:LIB$DT_STARTUP.COM
$ SHOW LOGICAL SYS$LANGUAGES
   "SYS$LANGUAGES" = "FRENCH" (LNM$SYSTEM_TABLE)
        = "GERMAN"
        = "SPANISH"
$ SHOW LOGICAL SYS$LANGUAGE
  "SYS$LANGUAGE" = "ENGLISH" (LNM$SYSTEM_TABLE)
```

If you do not define SYS$LANGUAGES, all translation routines default to English. See the OpenVMS run-time library documentation for more information on LIB$DT_STARTUP.COM.

The SET LANGUAGE statement does not affect the collating sequences used for sorting and comparing data. The CREATE COLLATING SEQUENCE statement specifies alternate collating sequences.

The SET LANGUAGE statement is available only on OpenVMS platforms. ♦

**LINE LENGTH n**
Specifies an alternate line length for SQL output.

You must enter a number *n* to designate the line length. The number *n* can be any number up to 512.

You can use the SET LINE LENGTH statement to specify an alternate width for output that you are sending to a file or to an alternate output device. For example, you could put two or more SET LINE LENGTH statements into a nested FOR loop to change the line length during the loop.

**OUTPUT file-spec**
Names the target file for output. The default file extension is .lis.

If you specify OUTPUT with a file name, SQL writes its output to a log file that you specify. The log file contains both statements and results. If you issue a SET OUTPUT statement, output is also written to standard output which is usually the terminal.

If you specify OUTPUT without a file name, SQL suspends writing output to a log file, if any, and writes the output to the standard output. In other words, the SET OUTPUT statement without a file name is equivalent to the SET NOOUTPUT statement.

SQL displays certain items (such as the headings produced by the SHOW statement) in boldface type on your terminal screen. In log files, however, the boldface items are surrounded by escape characters. You can ignore these escape characters, edit them out of your log file, or set your terminal so that SQL does not display characters in boldface type.

OpenVMS VAX OpenVMS Alpha On OpenVMS, if you disable boldface type using the following DCL command, your log file will not contain escape characters:

```
$ SET TERM/NOANSI_CRT
```

**NOOUTPUT**
Suspends writing to the output file.

**QUERY CONFIRM**
Lets you preview the cost of a query, in terms of I/O, before any rows are actually returned. For example:

```
SQL> SELECT * FROM EMPLOYEES;
Estimate of query cost:  52 I/O's, rows to deliver:  100
Do you wish to cancel this query (No)? YES
%SQL-F-QUERYCAN, Query cancelled at user's request
```

Some queries can result in Oracle Rdb performing a large number of I/O operations, retrieving a large number of rows, or both. The SET QUERY CONFIRM statement causes SQL to display estimated query costs. If the cost appears excessive, you can cancel the query by answering No; to continue, answer Yes.

**QUERY LIMIT**
Sets limits to restrict the output generated by a query.

The mechanism used to set these limits is called the query governor. The following gives you three ways to set limits using the query governor:

- ROWS total-rows

  You can restrict output by limiting the number of rows a query can return. The optimizer counts each row returned by the query and stops execution when the row limit is reached.

  The default is an unlimited number of row fetches. Dynamic SQL options are inherited from the compilation qualifier.

- TIME total-seconds

  You can restrict the amount of time used to optimize a query for execution. If the query is not optimized and prepared for execution before the total elapsed time limit is reached, an error message is returned.

  The default is unlimited time for the query compilation and execution. Dynamic SQL options are inherited from the compilation qualifier.

  **Note**

  Specifying a query time limit can cause application failure in certain circumstances. An application that runs successfully during off-peak hours may fail when run during peak hours.

- CPU TIME total-seconds

  You can restrict the amount of CPU time used to optimize a query for execution. If the query is not optimized and prepared for execution before the CPU time limit is reached, an error message is returned.

  The default is unlimited CPU time for the query compilation. Dynamic SQL options are inherited from the compilation qualifier.

Use a positive integer for the number of rows and the number of seconds; negative integers are invalid and zero means no limits. If an established limit is exceeded, the query is canceled and an error message is displayed. When you set both a time limit and the row limit, whichever value is reached first stops the output.

Application developers and 4GL tools can use this feature to prevent users from overloading the system. The database administrator can manage system performance and reduce unnecessary resource usage by setting option limits.

**RADIX POINT radix-char**
Changes the output displaying the radix point to the specified character. The radix point is the symbol that separates units from decimal fractions. For example, in the number 98.6, the period is the radix point.

You must enclose the radix-char argument within single quotation marks.

If you do not specify an alternate character, the default is either the period (.) or, on OpenVMS, the value specified by the logical name SYS$RADIX_POINT.

Digital UNIX

When you use Oracle Rdb for Digital UNIX, you must specify a radix character. ♦

**VERIFY**
Displays indirect command files at your terminal as you run them.

**NOVERIFY**
Does not display indirect command files. The default setting is the setting currently in effect for DCL commands. If you have not explicitly changed the DCL setting to VERIFY, the default is NOVERIFY.

**WARNING DEPRECATE**
**WARNING NODEPRECATE**
Specifies whether or not interactive SQL displays diagnostic messages when you issue statements containing obsolete SQL syntax. Deprecated or obsolete syntax is syntax that was allowed in previous versions of SQL but has been changed. Oracle Rdb recommends that you avoid using such syntax because it may not be supported in future versions. By default, SQL displays a warning

message after any statement containing obsolete syntax (SET WARNING DEPRECATE).

If you specify SET WARNING NODEPRECATE, SQL does not display any messages about obsolete syntax.

**DEFAULT CONSTRAINT MODE ON**
**DEFAULT CONSTRAINT MODE OFF**
Lets you change the default constraint setting for your session.

Before you execute the SET DEFAULT CONSTRAINT MODE statement, the default mode for constraints will be set at OFF. After you execute the SET DEFAULT CONSTRAINT MODE statement, any subsequent transactions have the initial constraint evaluation mode that you set. See the Examples to see how to use the SET statement to change the current setting for a constraint evaluation.

The SET ALL CONSTRAINTS ON statement causes all the affected constraints to be evaluated immediately, as well as at the end of each statement. Setting constraints OFF defers constraint evaluation until commit time. See the SET ALL CONSTRAINTS Statement for further information.

If the constraint setting is ON and the default constraint mode is ON, the constraint mode remains ON after commit or rollback operations. If the default constraint mode is ON, it remains ON until you issue a SET DEFAULT CONSTRAINT MODE OFF statement or exit from interactive SQL. A DISCONNECT DEFAULT statement will not set the default back to OFF.

If you issue a SET ALL CONSTRAINTS OFF statement while the default constraint setting is ON, the constraint mode is set to OFF for subsequent statements in that transaction, but the constraint mode reverts back to the default setting (ON) after a commit or rollback operation completes the current transaction.

**FLAGGER ON**
**FLAGGER SQL89**
**FLAGGER SQL92_ENTRY**
**FLAGGER MIA**
Controls the output of informational messages that indicate nonstandard syntax, that is, extensions to the ANSI/ISO standard syntax or the MIA standard syntax.

If you specify SET FLAGGER ON, which is the same as specifying SET FLAGGER SQL92_ENTRY ON, SQL sends you an informational message if you issue a subsequent interactive SQL statement that contains syntax that is an extension to the ANSI/ISO standard.

If you specify SET FLAGGER MIA ON, SQL sends you an informational message if you issue a subsequent interactive SQL statement that contains syntax that is an extension to the MIA standard.

The flaggers are independent of each other and any combination of flaggers can be set at one time.

The default is FLAGGER OFF if you do not explicitly set a flagger on.

**FLAGGER OFF**
Disables all previously set flaggers indicating nonstandard syntax. This is the default.

## Usage Notes

OpenVMS VAX OpenVMS Alpha

- The SET LANGUAGE statement does not affect the collating sequences used for sorting and comparing data. The CREATE COLLATING SEQUENCE statement specifies alternate collating sequences. ♦

OpenVMS VAX OpenVMS Alpha

- You cannot use the SET LANGUAGE statement in dynamic SQL; instead, you should use the logical name SYS$LANGUAGE as documented in Table 7–4. ♦
- The SET RADIX POINT statement changes the radix point only in the output display. It does not change the input character; the input character must always be a period.
- The SET DIGIT SEPARATOR statement changes the digit separator only in the output display. You cannot use a digit separator when inserting data.
- The alternate date and time formats allowed by the SET DATE FORMAT statement affect only date string text literals and their conversion to and from binary dates.
- The SET DATE FORMAT statement will not override input and output formats that you specified using an edit string. For example:

```
SQL> --
SQL> ALTER TABLE EMPLOYEES
cont> ALTER BIRTHDAY EDIT STRING "NN/DD/YYYY";
SQL> SELECT BIRTHDAY FROM EMPLOYEES;
 BIRTHDAY
  3/28/1947
  5/15/1954
  3/20/1954
  3/05/1937
        .
        .
        .
SQL> --
SET DATE FORMAT DATE 6
SQL> --
SQL> -- You might expect the dates to appear as 28 MAR 47, and
SQL> -- so on, but the SET DATE FORMAT statement does not
SQL> -- override the edit string:
SQL> --
SQL> SELECT BIRTHDAY FROM EMPLOYEES;
 BIRTHDAY
  3/28/1947
  5/15/1954
  3/20/1954
  3/05/1937
        .
        .
        .
SQL> ROLLBACK;
```

- To produce the default currency indicator or digit separator, you must specify an edit string for that column. If you subsequently set the currency indicator or digit separator to be a different character and then roll back the transaction, the output will not display the currency indicator or digit separator. This is because the ROLLBACK statement has rolled back the ALTER statement that specified the edit string. The SET CURRENCY SIGN or SET DIGIT SEPARATOR statement is still in effect for the remainder of the session. See Example 2.

OpenVMS VAX OpenVMS Alpha

- Table 7–4 lists the logical names you can use to internationalize the SET statement. You can specify the currency sign, date and time output format, digit separator, language, and radix point.

**Table 7–4 Logical Names for Internationalization of SET Statements**

| SQL SET Statement | Related System Logical Name |
|---|---|
| CURRENCY SIGN | SYS$CURRENCY |
| DATE FORMAT DATE date-number | LIB$DT_FORMAT |
| DATE FORMAT TIME time-number | LIB$DT_FORMAT |
| DIGIT SEPARATOR | SYS$DIGIT_SEP |
| LANGUAGE | SYS$LANGUAGE |
| RADIX POINT | SYS$RADIX_POINT |

If you want to change the input format for dates and time, you must use the logical name LIB$DT_INPUT_FORMAT documented in the OpenVMS run-time library documentation. The SET DATE FORMAT DATE and SET DATE FORMAT TIME statements in SQL change only the date and time formats for output displays. ♦

- Oracle Rdb is currently in compliance with the entry-level ANSI/ISO SQL standard.
- The SET FLAGGER ON statement is equivalent to the SET FLAGGER SQL92_ENTRY ON statement.
- You can set flaggers on and off independent of each other. For example:

```
SQL> SHOW FLAGGER
The flagger mode is OFF
SQL> --
SQL> SET FLAGGER SQL89 ON;
SQL> SHOW FLAGGER
%SQL-I-NONSTASYN89, Nonstandard SQL89 syntax
The SQL89 flagger mode is ON
SQL> --
SQL> SET FLAGGER MIA ON;
%SQL-I-NONSTASYN89, Nonstandard SQL89 syntax
SQL> SHOW FLAGGER
%SQL-I-NONSTASYN89, Nonstandard SQL89 syntax
The SQL89 flagger mode is ON
The MIA flagger mode is ON
SQL> --
SQL> SET FLAGGER SQL92_ENTRY ON;
%SQL-I-NONSTASYN, Nonstandard syntax
%SQL-I-NONSTASYN89, Nonstandard SQL89 syntax
SQL> SHOW FLAGGER
%SQL-I-NONSTASYN89, Nonstandard SQL89 syntax
%SQL-I-NONSTASYN92E, Nonstandard SQL92 Entry-level syntax
The SQL89 flagger mode is ON
The SQL92 Entry-level flagger mode is ON
The MIA flagger mode is ON
SQL> --
SQL> SET FLAGGER SQL89 OFF;
%SQL-I-NONSTASYN, Nonstandard syntax
%SQL-I-NONSTASYN89, Nonstandard SQL89 syntax
%SQL-I-NONSTASYN92E, Nonstandard SQL92 Entry-level syntax
SQL> SHOW FLAGGER;
%SQL-I-NONSTASYN92E, Nonstandard SQL92 Entry-level syntax
The SQL92 Entry-level flagger mode is ON
The MIA flagger mode is ON
```

- You cannot redefine standard output to redirect output to a file. Use the SET OUTPUT statement to redirect the output to a file.

## Examples

Example 1: Using the SET statement to set up terminal session characteristics

Using the SET statement as follows, you can set up the characteristics of your terminal session:

```
SQL> --
SQL> -- You can put the SET statements in your sqlini file, which sets up
SQL> -- your SQL session.
SQL> --
SQL> SET OUTPUT 'LOG.LIS'
SQL> SET DICTIONARY 'CDD$TOP.DEPT3'
SQL> SET EDIT KEEP 10
SQL> --
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> SHOW ALIAS
Alias PERS:
    Rdb database in file personnel
SQL> EXIT
```

In the preceding example, the statements set up the characteristics, as follows:

- The SET OUTPUT statement opens a file called LOG.LIS in the current default path name. From this point on, all the input and output, including error messages, appear in this file. The following example shows what is written to the log file LOG.LIS:

```
SET DICTIONARY 'CDD$TOP.DEPT3'
SET EDIT KEEP 10
--
ATTACH 'ALIAS PERS FILENAME personnel';
SHOW ALIAS
Alias PERS:
    Rdb database in file personnel
EXIT
```

OpenVMS VAX OpenVMS Alpha
- The SET DICTIONARY statement changes the default repository path name. ♦

OpenVMS VAX OpenVMS Alpha
- The SET EDIT KEEP statement specifies that you get the 10 previous statements in the editing buffer when you type EDIT *. ♦
- The ATTACH statement attaches to the personnel database and declares the alias PERS for that database.
- The SHOW ALIAS statements tell the user which alias is declared.

Example 2: SET CURRENCY SIGN and SET DIGIT SEPARATOR statements with the ROLLBACK statement

The following example uses the SET DIGIT SEPARATOR statement to show the behavior of the SET CURRENCY SIGN and SET DIGIT SEPARATOR statements when used with edit strings and the ROLLBACK statement:

```
SQL> --
SQL> -- This example shows the edit string "ZZZ,ZZZ",
SQL> -- which specifies the comma as the default digit separator.
SQL> --
SQL> ALTER TABLE SALARY_HISTORY -
cont> ALTER SALARY_AMOUNT EDIT STRING "ZZZ,ZZZ";
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
 SALARY_AMOUNT
        26,291
        51,712
        26,291
        50,000
        .
        .
        .
```

```
SQL> --
SQL> -- Now use the SET DIGIT SEPARATOR statement to specify that
SQL> -- the period will be the digit separator instead of
SQL> -- the comma.
SQL> --
SQL> SET DIGIT SEPARATOR "."
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
 SALARY_AMOUNT
        26.291
        51.712
        26.291
        50.000
        .
        .
        .
SQL> --
SQL> -- Roll back all changes made to this point.
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> -- The output now shows no digit separator.  This is because
SQL> -- the ALTER statement containing the edit string was
SQL> -- rolled back.
SQL> --
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
  SALARY_AMOUNT
       26291.00
       51712.00
       26291.00
       50000.00
        .
        .
        .
SQL> --
SQL> -- If you again specify an edit string, SQL again displays
SQL> -- the digit separator you specified with the SET DIGIT
SQL> -- SEPARATOR statement.
SQL> --
SQL> ALTER TABLE SALARY_HISTORY -
cont> ALTER SALARY_AMOUNT EDIT STRING "ZZZ,ZZZ";
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
 SALARY_AMOUNT
        26.291
        51.712
        26.291
        50.000
SQL> --
SQL> -- Roll back these changes.
SQL> ROLLBACK;
```

Example 3: Using the internationalization features of the SET statement

The following example shows how to use the various SET statements to internationalize your applications:

```
SQL> --
SQL> -- This first statement specifies the dollar sign
SQL> -- as the currency indicator. It does this by using
SQL> -- the edit string '$(9).99'.
SQL> --
SQL> ALTER TABLE SALARY_HISTORY -
cont> ALTER SALARY_AMOUNT EDIT STRING '$(9).99';
cont> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
  SALARY_AMOUNT
     $26291.00
     $51712.00
     $26291.00
     $50000.00
        .
        .
        .
SQL> --
SQL> -- The SET CURRENCY statement now changes the currency
SQL> -- indicator to the British pound sign, £.  Notice
SQL> -- the changed output.
SQL> --
SQL> SET CURRENCY SIGN '£'
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
 SALARY_AMOUNT
     £26291.00
     £51712.00
     £26291.00
     £50000.00
     £11676.00
        .
        .
        .
SQL> --
SQL> -- The next examples show the SET DATE FORMAT statement.
SQL> --
SQL> -- The SET DATE FORMAT statement will not override input
SQL> -- and output formats that you have specified with an edit
SQL> -- string.  The following SET DATE FORMAT examples use the
SQL> -- SALARY_START and SALARY_END columns.  The SALARY_START
SQL> -- and SALARY_END columns are defined by the domain
SQL> -- DATE_DOM, which uses the edit string 'DD-MMM-YYY'.
SQL> -- Thus, to test the SET DATE FORMAT statement, you must
SQL> -- first remove the edit string from the DATE_DOM domain
SQL> -- using the following ALTER DOMAIN statement:
SQL> --
SQL> ALTER DOMAIN DATE_DOM NO EDIT STRING;
SQL> --
SQL> -- The next statement inserts a row with time information.
SQL> -- The subsequent SET DATE FORMAT statements will use this row:
SQL> --
SQL> INSERT INTO SALARY_HISTORY
cont> -- list of columns:
cont> (EMPLOYEE_ID,
cont> SALARY_AMOUNT,
cont> SALARY_START,
cont> SALARY_END)
cont> VALUES
cont>  -- list of values:
cont> ('88339',
cont> '22550',
cont> '14-NOV-1967 08:30:00.00',
cont> '25-NOV-1988 16:30:00.00')
cont> ;
1 row inserted
SQL> --
SQL> -- Using the row that was just inserted, the following statement
SQL> -- shows the default date and time output:
SQL> --
SQL> SELECT SALARY_START, SALARY_END FROM SALARY_HISTORY-
cont> WHERE EMPLOYEE_ID = '88339';
  SALARY_START              SALARY_END
 14-NOV-1967 08:30:00.00   25-NOV-1988 16:30:00.00
1 row selected
SQL> --
SQL> -- The SET DATE FORMAT DATE statement customizes the
SQL> -- output of the date format.
SQL> --
SQL> -- The output will appear in the form
SQL> -- 14 NOV 67, as specified by the date-number argument 6.
SQL> --
SQL> SET DATE FORMAT DATE 6;
SQL> SELECT SALARY_START, SALARY_END FROM SALARY_HISTORY-
cont> WHERE EMPLOYEE_ID = '88339';
 SALARY_START   SALARY_END
 14 NOV 67      25 NOV 88
1 row selected
SQL> --
SQL> -- The SET DATE FORMAT TIME statement customizes
SQL> -- the output of the time format.  The output will appear
SQL> -- in the form 16 h 30 min 0 s, as specified by the
SQL> -- time-number argument 20.
SQL> --
SQL> SET DATE FORMAT TIME 20;
SQL> SELECT SALARY_START, SALARY_END FROM SALARY_HISTORY-
cont> WHERE EMPLOYEE_ID = '88339';
 SALARY_START       SALARY_END
 8 h 30 min 0 s     16 h 30 min 0 s
1 row selected
```

```
SQL> --
SQL> -- Note that the previous date example has deleted
SQL> -- the time output, and the previous time example has
SQL> -- deleted the date output.
SQL> --
SQL> -- If you want the display to continue to show
SQL> -- BOTH date and time, you must specify
SQL> -- both arguments with the SET DATE statement.
SQL> --
SQL> SET DATE FORMAT DATE 6, TIME 20;
SQL>  SELECT SALARY_START, SALARY_END FROM SALARY_HISTORY-
cont> WHERE EMPLOYEE_ID = '88339';
 SALARY_START                SALARY_END
 14 NOV 67 8 h 30 min 0 s     25 NOV 88 16 h 30 min 0 s
1 row selected
SQL> --
SQL> -- The next example changes the digit separator to a period and
SQL> -- the radix point to a comma:
SQL> --
SQL> ALTER TABLE SALARY_HISTORY -
cont> ALTER SALARY_AMOUNT EDIT STRING 'ZZZ,ZZZ.ZZ';
SQL> --
SQL> SET RADIX POINT ','
SQL> SET DIGIT SEPARATOR '.'
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
 SALARY_AMOUNT
     26.291,00
     51.712,00
     26.291,00
     50.000,00
        .
        .
        .
SQL> --
SQL> -- This example shows how you can use the SET LANGUAGE
SQL> -- statement to change the output of dates to a particular
SQL> -- language.  This example shows the default English first,
SQL> -- followed by French.
SQL> --
SQL> -- Note that the time format is still based on
SQL> -- the SET DATE FORMAT TIME statement
SQL> -- previously executed in this example.
SQL> --
SQL> SELECT SALARY_START FROM SALARY_HISTORY;
 SALARY_START
  5 JUL 80 0 h 0 min 0 s
 14 JAN 83 0 h 0 min 0 s
  2 MAR 81 0 h 0 min 0 s
 21 SEP 81 0 h 0 min 0 s
  3 NOV 81 0 h 0 min 0 s
  1 JUL 82 0 h 0 min 0 s
 27 JAN 81 0 h 0 min 0 s
  1 JUL 75 0 h 0 min 0 s
 29 DEC 78 0 h 0 min 0 s
  2 FEB 80 0 h 0 min 0 s
  8 APR 79 0 h 0 min 0 s
 19 AUG 77 0 h 0 min 0 s
        .
        .
        .
SQL> --
SQL> SET LANGUAGE FRENCH
SQL> SELECT SALARY_START FROM SALARY_HISTORY;
 SALARY_START
  5 jul 80 0 h 0 min 0 s
 14 jan 83 0 h 0 min 0 s
  2 mar 81 0 h 0 min 0 s
 21 sep 81 0 h 0 min 0 s
  3 nov 81 0 h 0 min 0 s
  1 jul 82 0 h 0 min 0 s
 27 jan 81 0 h 0 min 0 s
  1 jul 75 0 h 0 min 0 s
 29 déc 78 0 h 0 min 0 s
  2 fév 80 0 h 0 min 0 s
  8 avr 79 0 h 0 min 0 s
 19 aoû 77 0 h 0 min 0 s
        .
        .
        .
SQL> --
```

Example 4: Using the SET statement to change the current setting for constraint evaluation

The following example shows how to use the SET statement to change the constraint evaluation mode for the current transaction. You can display both the current setting and the default setting.

```
SQL> ATTACH 'FILENAME personnel';
SQL> --
SQL> -- Show settings before starting, set the default mode,
SQL> -- then show the settings again.
SQL> --
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is off
SQL> SET DEFAULT CONSTRAINT MODE ON;
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is on
SQL> --
SQL> -- This INSERT statement fails because the default
SQL> -- constraint mode is set to ON.
SQL> --
SQL> INSERT INTO DEGREES VALUES
cont>    ('00164','NSTU',1990,'MS','Applied Math');
%RDB-E-INTEG_FAIL, violation of constraint DEGREES_FOREIGN2 caused
operation to fail
-RDB-F-ON_DB, on database DISK2:[SMALLWOOD]PERSONNEL.RDB;1
SQL> --
SQL> -- Insert a valid record.
SQL> INSERT INTO DEGREES VALUES
cont>    ('00164','HVDU',1990,'MS','Comp. Science');
1 row inserted
SQL> --
SQL> -- This rollback will not turn off the default constraint mode.
SQL> --
SQL> ROLLBACK;
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is on
SQL> --
SQL> -- This SET ALL CONSTRAINTS OFF statement will cause any
SQL> -- constraints in the current transaction to be evaluated
SQL> -- at commit time.
SQL> --
SQL> SET ALL CONSTRAINTS OFF;
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is on
    Statement constraint evaluation is off
SQL> UPDATE DEGREES SET DEGREE = 'MEd' WHERE EMPLOYEE_ID ='00171';
2 rows updated
SQL> COMMIT;
%RDB-E-INTEG_FAIL, violation of constraint DEG_DEGREE_VALUES caused
operation to fail
-RDB-F-ON_DB, on database DISK2:[SMALLWOOD]PERSONNEL.RDB;1
SQL> --
SQL> SELECT * FROM DEGREES WHERE DEGREE = 'MEd';
 EMPLOYEE_ID   COLLEGE_CODE   YEAR_GIVEN   DEGREE   DEGREE_FIELD
 00171         QUIN                 1982   MEd      Applied Math
 00171        UME                1978   MEd      Elect. Engrg.
2 rows selected
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is on
    Statement constraint evaluation is off
SQL> --
SQL> -- This rollback will complete the transaction and the
SQL> -- constraint setting will be back to the default mode.
SQL> --
SQL> ROLLBACK;
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is on
SQL> --
SQL> -- Constraints will continue to be evaluated according to the
SQL> -- default mode until either a SET DEFAULT CONSTRAINT MODE OFF
SQL> -- or a SET ALL CONSTRAINTS OFF statement is issued.
SQL> --
SQL> INSERT INTO DEGREES VALUES ('00171','HVDU',1990,'MEd','Education');
%RDB-E-INTEG_FAIL, violation of constraint DEG_DEGREE_VALUES caused
operation to fail
-RDB-F-ON_DB, on database DISK2:[SMALLWOOD]PERSONNEL.RDB;1
SQL> --
SQL> SHOW CONSTRAINT MODE;
    Statement constraint evaluation default is on
    Statement constraint evaluation is on
SQL> ROLLBACK;
```

Example 5: Using the SET statement to send messages about syntax that contains extensions to the ANSI/ISO SQL or MIA standards

This example shows the output when flagging is turned on, first for SQL92_ ENTRY and then for MIA.

```
SQL> -- Flagging is off by default. When you enter a statement that
SQL> -- uses the data type VARCHAR, SQL does not issue a message.
SQL> --
SQL> SHOW FLAGGER MODE;
The flagger mode is OFF
SQL> CREATE TABLE TEST1 (TEXT_COL VARCHAR (100));
SQL> --
SQL> -- When you set the flagger to SQL92_ENTRY, SQL generates an
SQL> -- error message because VARCHAR is an extension to the standard.
SQL> --
SQL> SET FLAGGER SQL92_ENTRY ON
SQL> CREATE TABLE TEST2 (TEXT_COL VARCHAR (100));
%SQL-I-NONSTADTP, Nonstandard data type
SQL> --
SQL> -- With the flagger set to SQL92_ENTRY, SQL does not generate an
SQL> -- error message for the data type CHAR because it is an ANSI/ISO
SQL> -- standard data type.
SQL> --
SQL> CREATE TABLE TEST3 (TEXT_COL CHAR);
SQL> --
SQL> -- However, when you set the flagger to MIA, SQL generates two
SQL> -- error messages because data definition is not part of the MIA
SQL> -- standard. The first error message is caused by the CREATE
SQL> -- keyword; the second is caused by trying to create a table.
SQL> --
SQL> -- (Note that the SET FLAGGER statement itself is nonstandard.)
SQL> --
SQL> SET FLAGGER MIA ON
%SQL-I-NONSTASYN, Nonstandard syntax
SQL> CREATE TABLE TEST3 (TEXT_COL CHAR);
%SQL-I-NONSTASYN, Nonstandard syntax
%SQL-I-NONSTASYN, Nonstandard syntax
SQL>
```

Example 6: Using the SET statement to check for syntax errors before issuing a statement

This example shows the output from a statement when the user specifies EXECUTE and the output from the same statement when the user specifies NOEXECUTE, but commits a syntax error.

```
SQL> -- When the EXECUTE option is set, SQL displays output from
SQL> -- each statement normally.
SQL> --
SQL> SELECT E.LAST_NAME FROM EMPLOYEES E;
 LAST_NAME
 Ames
 Andriola
 Babbin
 Bartlett
   .
   .
   .
SQL> --
SQL> -- When the NOEXECUTE option is set, SQL does not display statement
SQL> -- output, but does return syntax error messages.
SQL> --
SQL> SET NOEXECUTE
SQL> SEKECT E.LAST_NAME FROM EMPLOYEES E;
SEKECT E.LAST_NAME FROM EMPLOYEES E;
^
%SQL-F-LOOK_FOR_STMT, Syntax error, looking for a valid SQL statement,
found SEKECT instead
SQL> --
SQL> -- After the spelling error is corrected, the statement is
SQL> -- error-free, but with NOEXECUTE specified, SQL does not
SQL> -- display output.
SQL> --
SQL> SELECT E.LAST_NAME FROM EMPLOYEES E;
SQL>
```

Example 7: Using the SET statement to check for obsolete syntax

This example shows the output from an obsolete SQL statement when the user specifies WARNING DEPRECATE, and the output from the same statement when the user specifies WARNING NODEPRECATE.

```
SQL> --
SQL> -- By default, SQL sends warning messages when you use obsolete syntax.
SQL> --
SQL> DECLARE SCHEMA FILENAME personnel;
%SQL-I-DEPR_FEATURE, Deprecated Feature: SCHEMA (meaning ALIAS)
SQL> --
SQL> -- When you specify SET WARNING NODEPRECATE, SQL does not display warning
SQL> -- messages.
SQL> --
SQL>DECLARE SCHEMA FILENAME personnel;
This alias has already been declared.
Would you like to override this declaration (No)?
SQL>
```

Example 8: Using the SET statement to specify the amount of CPU time to be expended on query compilation

```
SQL> SET QUERY LIMIT CPU TIME 10;
```

## SET ALIAS Statement

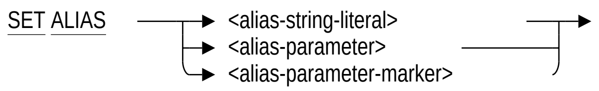
Specifies the default alias for an SQL user session in dynamically prepared and executed or interactive SQL until another SET ALIAS statement is issued. If you do not specify an alias, the default is RDB$DBHANDLE.

### Environment

You can use the SET ALIAS statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**alias-string-literal**
Specifies a character string literal that specifies the default alias. The alias string literal must be enclosed in single quotation marks.

**alias-parameter**
Specifies a host language variable in precompiled SQL or a formal parameter in an SQL module language procedure that specifies the default alias.

**alias-parameter-marker**
Specifies a parameter marker (?) in a dynamic SQL statement. The alias parameter marker refers to a parameter that specifies the default alias.

## Usage Notes

- SQL interprets a two-level name in the following way:
  1. SQL checks the name to the left of the period (.) to determine if it is an alias. If it is, SQL interprets the name as:

     ```
     alias-name.table-name
     ```

  2. If there is no alias for this name, then SQL interprets the two-level name as:

     ```
     schema-name.table-name
     ```

## Examples

Example 1: Setting a default alias to avoid qualifying object names

```
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> SET CATALOG 'ADMINISTRATION';
SQL> SET SCHEMA 'PERSONNEL';
SQL> SELECT LAST_NAME FROM EMPLOYEES;
%SQL-F-NODEFDB, There is no default database
SQL> --
SQL> -- You must qualify the table name because you attached with an alias.
SQL> --
SQL> SELECT LAST_NAME FROM CORP.EMPLOYEES;
 LAST_NAME
 Ames
 Andriola
 Babbin
    .
    .
    .
100 rows selected
SQL> SET ALIAS 'CORP';
SQL> --
SQL> -- Now you do not need to qualify the table name EMPLOYEES.
SQL> --
SQL> SELECT LAST_NAME FROM EMPLOYEES;
 LAST_NAME
 Ames
 Andriola
 Babbin
    .
    .
    .
100 rows selected
```

Example 2: Changing the default alias

Use the SHOW DATABASE statement to see the database settings.

```
SQL> ATTACH 'FILENAME personnel';
SQL> --
SQL> ATTACH 'FILENAME corporate_data';
This alias has already been declared.
Would you like to override this declaration (No)? Y
SQL> --
SQL> -- There can be only one default alias.  The default alias,
SQL> -- RDB$DBHANDLE, is now assigned to corporate_data.
SQL> --
SQL> -- To set the default to a different database, change the default
SQL> -- alias before attaching to a database.  The next database you
SQL> -- attach to without an alias is given the new default alias
SQL> -- instead of RDB$DBHANDLE.
SQL> --
SQL> SET ALIAS 'CORP';
SQL> ATTACH 'FILENAME personnel';
```

## SET ALL CONSTRAINTS Statement

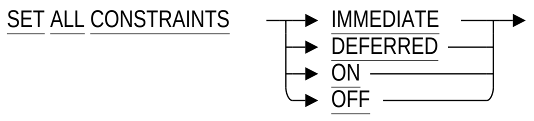
Controls checking for constraints that are evaluated at commit time. (This statement has no effect on constraints that are evaluated at verb time. For verb-time evaluation information, see the SET TRANSACTION Statement.) A major use of the SET ALL CONSTRAINTS statement is to permit evaluation of these constraints at intervals before the transaction is committed.

### Environment

You can use the SET ALL CONSTRAINTS statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**IMMEDIATE**
**DEFERRED**
Sets the constraint mode. The default constraint mode setting is DEFERRED, which means that constraint evaluation is deferred until commit time unless you have used one of the following to specify otherwise:

- SET DEFAULT CONSTRAINT MODE ON statement
- SQLOPTIONS=(CONSTRAINTS=ON) qualifier on the SQL precompiler command line
- CONSTRAINTS=ON qualifier on the SQL module language command line

When you issue a SET ALL CONSTRAINTS IMMEDIATE statement, SQL:

- Evaluates all previously deferred constraints (those that would otherwise be evaluated at a COMMIT statement)

## SET ALL CONSTRAINTS Statement

- Sets a mode in which SQL evaluates any constraints selected for deferred evaluation by the execution of an SQL statement at the end of that SQL statement (instead of waiting for a COMMIT statement)

Once the transaction completes, the constraint mode is set back to the default constraint mode for subsequent statements.

The SET ALL CONSTRAINTS DEFERRED statement causes constraint evaluation to be deferred until commit time, when the transaction completes.

The SET ALL CONSTRAINTS IMMEDIATE and SET ALL CONSTRAINTS DEFERRED statements comply with the ANSI/ISO SQL 1992 standard.

**ON**
**OFF**
Sets the constraint mode. The SET ALL CONSTRAINTS ON statement is equivalent to SET ALL CONSTRAINTS IMMEDIATE, and SET ALL CONSTRAINTS OFF is equivalent to SET ALL CONSTRAINTS DEFERRED. The ON and OFF keywords comply with the ANSI/ISO 1989 SQL standard; IMMEDIATE and DEFERRED comply with the ANSI/ISO SQL 1992 standard.

## Usage Notes

- If a transaction was declared but is not active when the SET ALL CONSTRAINTS statement is executed, SQL starts the declared transaction.
- See the description of the DEFAULT CONSTRAINT MODE argument in the SET Statement for more information about how the default constraint mode affects the SET ALL CONSTRAINTS statement.
- See the description of the SQLOPTIONS=(CONSTRAINTS=ON | OFF) qualifiers for the SQL precompiler command line in Chapter 4 and the CONSTRAINTS qualifier for the SQL module language command line in Chapter 3.
- If you require verb-time constraint evaluation, you must use the EVALUATING clause on the SQL SET TRANSACTION statement. The SQL SET ALL CONSTRAINTS statement only affects when deferred (commit time) constraints get evaluated. For information about the VERB TIME clause, see the SET TRANSACTION Statement.
- See the *Oracle Rdb7 Guide to SQL Programming* for information on guidelines for controlling constraint evaluation time.

## Example

Example 1: Using the SET ALL CONSTRAINTS statement in interactive SQL

```
SQL> SET ALL CONSTRAINTS IMMEDIATE;
SQL> SHOW CONSTRAINTS;
    Statement constraint evaluation default is off
    Statement constraint evaluation is on
SQL> -- Show the constraints
SQL> SHOW TABLES (CONSTRAINTS) JOB_HISTORY;

Table constraints for JOB_HISTORY:
JOB_HISTORY_FOREIGN1
 Foreign Key constraint
 Column constraint for JOB_HISTORY.EMPLOYEE_ID
 Evaluated on COMMIT
 Source:
        JOB_HISTORY.EMPLOYEE_ID REFERENCES EMPLOYEES (EMPLOYEE_ID)


JOB_HISTORY_FOREIGN2
 Foreign Key constraint
 Column constraint for JOB_HISTORY.JOB_CODE
 Evaluated on COMMIT
 Source:
        JOB_HISTORY.JOB_CODE REFERENCES JOBS (JOB_CODE)

JOB_HISTORY_FOREIGN3
 Foreign Key constraint
 Column constraint for JOB_HISTORY.DEPARTMENT_CODE
 Evaluated on COMMIT
 Source:
        JOB_HISTORY.DEPARTMENT_CODE REFERENCES DEPARTMENTS (DEPARTMENT_CODE)

Constraints referencing table JOB_HISTORY:
No constraints found

SQL>
SQL> SET ALL CONSTRAINTS DEFERRED;
SQL> SHOW CONSTRAINTS;
    Statement constraint evaluation default is off
    Statement constraint evaluation is off
```

## SET ANSI Statement

Specifies whether or not SQL behavior in certain instances complies with the ANSI/ISO SQL standard. The current default behavior in these instances is noncompliant.

**Note**

SQL provides the following new statements to replace the SET ANSI statement:

- SET DEFAULT DATE FORMAT replaces SET ANSI DATE; see the SET DEFAULT DATE FORMAT Statement.
- SET KEYWORD RULES replaces SET ANSI IDENTIFIERS; see the SET KEYWORD RULES Statement.
- SET QUOTING RULES replaces SET ANSI QUOTING; see the SET QUOTING RULES Statement.
- SET VIEW UPDATE RULES is new; see the SET VIEW UPDATE RULES Statement.

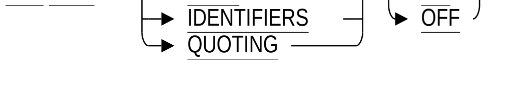
In addition, SQL provides the SET DIALECT statement to let you specify, with one statement, settings for all of these statements. See the SET DIALECT Statement for more information.

SQL does not return a deprecated feature message if you use the SET ANSI statement.

### Environment

You can use the SET ANSI statement only in interactive SQL.

### Format



## Arguments

**DATE ON**
**DATE OFF**
Specifies the default interpretation for columns with the DATE or CURRENT_TIMESTAMP data type.

The DATE and CURRENT_TIMESTAMP data types, can be either VMS ADT or ANSI. By default, both data types are interpreted as VMS ADT. The VMS ADT format contains YEAR TO SECOND fields, just as a TIMESTAMP does.

You can change DATE and CURRENT_TIMESTAMP to ANSI format with the SET DEFAULT DATE FORMAT statement, the precompiler DEFAULT DATE FORMAT clause in a DECLARE MODULE statement embedded in a program, or the module language DEFAULT DATE FORMAT clause in a module file. The ANSI format DATE contains only the YEAR TO DAY fields.

You must use the SET DEFAULT DATE FORMAT statement before creating domains or tables. You cannot use this statement to modify the data type once you have created a column or table.

**IDENTIFIERS ON**
**IDENTIFIERS OFF**
Specifies whether or not SQL checks statements that use reserved words as identifiers. If you specify SET ANSI IDENTIFIERS ON, SQL checks statements for reserved words from the ANSI/ISO 1989 standard. You must enclose reserved words in double quotation marks to supply them as identifiers in SQL statements. If you do not, SQL issues an informational message after such statements when you enable reserved-word checking. For a list of the reserved words deprecated as identifiers, see Section F.4.

When you specify SET ANSI IDENTIFIERS OFF, SQL does not check identifiers. By default, SQL does not check identifiers.

**QUOTING ON**
**QUOTING OFF**
Allows you to use double quotation marks to delimit the alias and catalog name pair in subsequent statements. By default, SQL syntax allows only single quotation marks. To comply with ANSI/ISO SQL standard naming conventions, ANSI QUOTING must be on. You must set ANSI QUOTING on to use multischema database naming.

## Example

Example 1: Setting CURRENT_TIMESTAMP to ANSI format

In the following example, SQL issues an error message because CURRENT_TIMESTAMP is an ADT data type by default, and TIMESTAMP is an ANSI data type. The SET ANSI DATE ON statement changes the default CURRENT_TIMESTAMP to ANSI format.

```
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
%SQL-F-DEFVALINC, You specified a default value for LOGGING_DATE which is
inconsistent with its data type
SQL> SET ANSI DATE ON;
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
SQL>
```

Example 2: Using the SET ANSI IDENTIFIERS statement to check for reserved words

This example shows the output from an SQL statement that creates a domain and specifies the ANSI89 reserved word CONTINUE as the user-supplied name for that domain. The SET ANSI IDENTIFIERS ON statement requires that you use uppercase characters for the name and enclose it in double quotation marks.

```
SQL> SET ANSI IDENTIFIERS ON;
SQL> CREATE DOMAIN CONTINUE CHAR (5);
%SQL-F-RES_WORD_AS_IDE, Keyword continue used as an identifier
SQL> CREATE DOMAIN "CONTINUE" CHAR (5);
```

## SET CATALOG Statement

Specifies the default catalog name for an SQL user session in dynamically prepared and executed or interactive SQL until another SET CATALOG statement is issued.

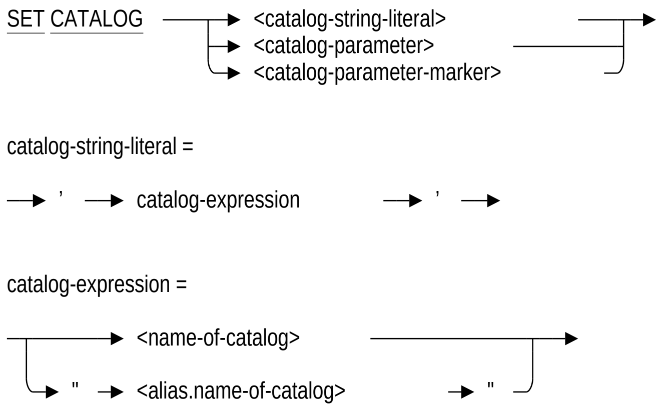
Within one multischema database, tables in different catalogs can be used in a single SQL statement; tables in catalogs in different databases cannot. If you omit the catalog name when you specify an object in a multischema database, SQL uses the default catalog name.

### Environment

You can use the SET CATALOG statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



## Arguments

**catalog-string-literal**
Specifies a character string literal that specifies the default catalog. The catalog string literal must contain a catalog expression enclosed in single quotation marks.

**catalog-parameter**
Specifies a host language variable in precompiled SQL or a formal parameter in an SQL module language procedure that specifies the default catalog. The catalog parameter must contain a catalog expression.

**catalog-parameter-marker**
Specifies a parameter marker (?) in a dynamic SQL statement. The catalog parameter marker refers to a parameter that specifies the default catalog. The catalog parameter marker must specify a parameter that contains a catalog expression.

**catalog-expression**
Specifies the name of the default catalog for a multischema database. If you omit the catalog name when you specify an object in a multischema database, SQL uses the default catalog name. If you do not specify a default catalog name, the default is RDB$CATALOG.

If you qualify the catalog name with an alias, the alias and catalog name pair must be in uppercase characters and you must enclose the alias and catalog name pair within double quotation marks.

See Section 2.2.7 for more information on catalogs.

## Usage Notes

- SQL does not issue an error message when you use SET CATALOG to set default to a catalog that does not exist. However, when you refer to that catalog by specifying an unqualified name, SQL issues the error message shown in the following example:

```
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> SHOW CATALOGS
Catalogs in database CORP
    "CORP.ADMINISTRATION"
    "CORP.RDB$CATALOG"
SQL> SET CATALOG '"CORP.NONEXISTENT"';
SQL> SET SCHEMA 'PERSONNEL';
SQL> CREATE TABLE NEWTABLE (COL1 REAL);
%SQL-F-CATNOTDEF, Catalog NONEXISTENT is not defined
```

- Remember that the double-quoted leftmost pair (the delimited identifier) in a multischema object name requires uppercase characters. For other multischema naming rules, see Section 2.2.3. You will receive the following error message if you specify a delimited identifier in lowercase characters:

```
SQL> SET SCHEMA '"corp.administration".accounting';
SQL> CREATE TABLE NEWTABLE (COL1 REAL);
%SQL-F-NODEFDB, There is no default database
SQL> SET SCHEMA '"CORP.ADMINISTRATION".accounting';
SQL> CREATE TABLE NEWTABLE (COL1 REAL);
SQL>
```

## Examples

Example 1: Setting schema and catalog defaults for the default database

In this example, the user attaches to the multischema corporate_data database, uses SET SCHEMA and SET CATALOG statements to change the defaults to catalog ADMINISTRATION and schema ACCOUNTING of the corporate_data database, and creates the table BUDGET in the schema ACCOUNTING.

```
SQL> ATTACH 'FILENAME corporate_data';
SQL> SHOW CATALOGS;
Catalogs in database with filename corporate_data
    ADMINISTRATION
    RDB$CATALOG
```

```
SQL>  SHOW SCHEMAS;
Schemas in database with filename corporate_data
    ADMINISTRATION.ACCOUNTING
    ADMINISTRATION.PERSONNEL
    ADMINISTRATION.RECRUITING
    RDB$SCHEMA
SQL> SET CATALOG 'ADMINISTRATION';
SQL> SET SCHEMA 'ACCOUNTING';
SQL> CREATE TABLE BUDGET (COL1 REAL);
SQL> SHOW TABLES;
     BUDGET
     DAILY_HOURS
     DEPARTMENTS
     .
     .
     .
SQL> --
SQL> -- To see the qualified table names, set default
SQL> -- to another schema and catalog.
SQL> --
SQL> SET CATALOG 'RDB$CATALOG';
SQL> SET SCHEMA 'RDB$SCHEMA';
SQL> SHOW TABLES
User tables in database with filename corporate_data
     ADMINISTRATION.ACCOUNTING.BUDGET
     ADMINISTRATION.ACCOUNTING.DAILY_HOURS
     ADMINISTRATION.ACCOUNTING.DEPARTMENTS
     .
     .
     .
```

Example 2: Setting a default catalog for a database with an alias

In this example, the user attaches to the multischema corporate_data database using the alias CORP. Setting the default catalog allows you to shorten the table name because you can qualify it with just the schema.

```
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> CREATE TABLE ACCOUNTING.PROJECT_7 (STATUS REAL);
%SQL-F-DBHANDUNK, ACCOUNTING is not the alias of a known database
SQL> --
SQL> -- You cannot qualify the table name without the alias,
SQL> -- so SQL assumes ACCOUNTING is the alias, not the schema.
SQL> -- Unless you want to qualify the table name with
SQL> -- both alias and catalog names, you must set the
SQL> -- default catalog to ADMINISTRATION, which
SQL> -- contains ACCOUNTING. You must enable ANSI/ISO quoting to do this.
SQL> --
SQL> SET QUOTING RULES 'SQL92';
SQL> SET CATALOG '"CORP.ADMINISTRATION"';
SQL> CREATE TABLE ACCOUNTING.PROJECT_7 (STATUS REAL);
SQL> SHOW TABLES;
User tables in database with filename corporate_data
     ACCOUNTING.BUDGET
   .
   .
   .
     ACCOUNTING.PROJECT_7
     ACCOUNTING.WORK_STATUS
   .
   .
   .
```

## SET CHARACTER LENGTH Statement

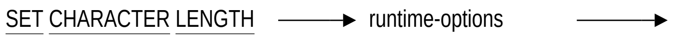
Specifies whether the length of character string parameters, columns, domains, and offsets are interpreted as characters or octets. (An **octet** is a group of 8 bits.)

### Environment

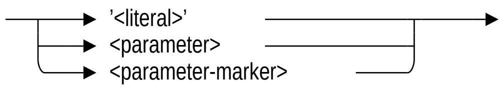
You can use the SET CHARACTER LENGTH statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**′literal′**
**parameter**
**parameter-marker**
Specifies the value of runtime-options, which must be one of the following:

- OCTETS
- CHARACTERS

**OCTETS**
Specifies the length of character string parameters, columns, domains, and offsets, which are interpreted as octets.

The default is octets.

**CHARACTERS**
Specifies the length of character string parameters, columns, domains, and offsets, which are interpreted as characters.

## Usage Notes

- If the SET DIALECT statement is processed after the SET CHARACTER LENGTH statement, it can override the setting of the SET CHARACTER LENGTH statement.
- If the CHARACTER LENGTH is set to OCTETS and you use a multi-octet character set, you must specify an appropriate size for parameters, columns, and domains.
- Use the SHOW CONNECTIONS CURRENT statement to see the current setting of character length for the session.

## Examples

Example 1: Setting the character length to octets

```
SQL> SET CHARACTER LENGTH 'OCTETS';
SQL> --
SQL> SHOW CONNECTIONS CURRENT;
Connection: -default-
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: SQLV40
Default DATE type: DATE VMS
Quoting Rules: SQLV40
Optimization Level: DEFAULT
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

```
Alias RDB$DBHANDLE:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
SQL> --
SQL> -- Create two domains: one uses DEC_MCS, a single-octet character
SQL> -- set, and one uses KANJI, a fixed multi-octet character set.
SQL> --
SQL> SET CHAR LENGTH 'OCTETS';
SQL> CREATE DOMAIN MCS_DOM CHAR(8) CHARACTER SET DEC_MCS;
SQL> CREATE DOMAIN KANJI_DOM CHAR(5) CHARACTER SET KANJI;
%SQL-F-CHRUNIBAD, Number of octets is not an integral number of characters
SQL> --
SQL> -- Because KANJI is a fixed multi-octet character set, using two
SQL> -- octets for each character, you must specify the size as a multiple
SQL> -- of two.
SQL> --
SQL> CREATE DOMAIN KANJI_DOM CHAR(8) CHARACTER SET KANJI;
SQL> SHOW DOMAINS;
User domains in database with filename MIA_CHAR_SET
KANJI_DOM                       CHAR(8)
         KANJI 4 Characters,  8 Octets
MCS_DOM                         CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
```

Example 2: Setting the character length to characters

```
SQL> SET CHARACTER LENGTH 'CHARACTERS';
SQL> SHOW CONNECTIONS CURRENT;
Connection: -default-
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: CHARACTERS
Keyword Rules: SQLV40
View Rules: SQLV40
Default DATE type: DATE VMS
Quoting Rules: SQLV40
Optimization Level: DEFAULT
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

```
Alias RDB$DBHANDLE:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
SQL> --
SQL> -- Create two domains: one uses DEC_MCS, a single-octet character
SQL> -- set, and one uses KANJI, a fixed multi-octet character set.
SQL> --
SQL> CREATE DOMAIN MCS_DOM CHAR(8) CHARACTER SET DEC_MCS;
SQL> CREATE DOMAIN KANJI_DOM CHAR(8) CHARACTER SET KANJI;
SQL> SHOW DOMAINS;
User domains in database with filename MIA_CHAR_SET
KANJI_DOM                       CHAR(8)
         KANJI 8 Characters,  16 Octets
MCS_DOM                         CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
```

## SET CONNECT Statement

Selects the named connection from the available connections, suspends any current connection and saves its context, and uses the named connection in subsequent procedures in the application after the SET CONNECT statement executes.

When SQL executes the SET CONNECT statement, the connection that you specify is used by subsequent procedures in the application.

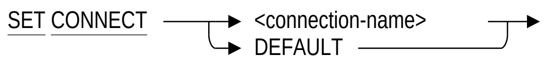
For information about creating and naming connections, see the CONNECT Statement.

### Environment

You can use the SET CONNECT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**connection-name**
Specifies a name for the association between the group of databases being attached (the environment) and the database and request handles that reference them (the connection).

You can specify the connection name as the following:

- String literal enclosed within single quotation marks
- Parameter (in module language)
- Variable (in precompiled SQL)

**DEFAULT**
Specifies one or more databases to be attached as a unit.

Use the DEFAULT keyword to specify the default connection. The default connection is all the databases that were attached interactively, or all those made known to the module at compile time through DECLARE ALIAS statements.

## Usage Note

If you specify a connection name unknown to SQL, SQL returns an error message and does not change the connection state. SQL suspends the current connection and saves all context.

## Examples

Example 1: Creating a default connection and two other connections

The following log file from an interactive SQL connection shows three databases attachments: personnel_northwest, personnel_northeast, and personnel_southeast. (By not specifying an alias for personnel_northwest, the default alias is assigned.) Several connections are established, including EAST_COAST, which includes both personnel_northeast and personnel_southeast.

Use the SHOW DATABASE statement to see the database settings.

```
SQL> --
SQL> -- Attach to the personnel_northwest and personnel_northeast databases.
SQL> -- personnel_northwest has the default alias, so personnel_northeast
SQL> -- requires an alias.
SQL> -- All the attached databases comprise the default connection.
SQL> --
SQL> ATTACH 'FILENAME personnel_northwest';
SQL> ATTACH 'ALIAS NORTHEAST FILENAME personnel_northeast';
SQL> --
SQL> -- Add the personnel_southeast database.
SQL> --
SQL> ATTACH 'ALIAS SOUTHEAST FILENAME personnel_southeast';
SQL> --
SQL> -- Connect to personnel_southeast.  CONNECT does an
SQL> -- implicit SET CONNECT to the newly created connection.
SQL> --
SQL> CONNECT TO 'ALIAS SOUTHEAST FILENAME personnel_southeast'
cont>     AS 'SOUTHEAST_CONNECTION';
SQL> --
SQL> -- Connect to both personnel_southeast and personnel_northeast as
SQL> -- EAST_COAST connection. SQL replaces the current connection to
SQL> -- the personnel_southeast database with the EAST_COAST connection
SQL> -- when you issue the CONNECT statement. You now have two different
SQL> -- connections that include personnel_southeast.
SQL> --
SQL> CONNECT TO 'ALIAS NORTHEAST FILENAME personnel_northeast,
cont>     ALIAS SOUTHEAST FILENAME personnel_southeast'
cont>     AS 'EAST_COAST';
SQL> --
SQL> -- The DEFAULT connection still includes all the attached databases.
SQL> --
SQL> SET CONNECT DEFAULT;
SQL> --
SQL> -- DISCONNECT releases the connection name EAST_COAST, but
SQL> -- does not detach from the EAST_COAST databases because
SQL> -- they are also part of the default connection.
SQL> --
SQL> DISCONNECT 'EAST_COAST';
SQL> --
SQL> SET CONNECT 'EAST_COAST';
%SQL-F-NOSUCHCON, There is not an active connection by that name
SQL> --
SQL> -- If you disconnect from the default connection, and have no other
SQL> -- current connections, you are no longer attached to any databases.
SQL> --
SQL> DISCONNECT DEFAULT;
SQL> SHOW DATABASES;
%SQL-F-ERRATTDEF, Could not use database file specified by SQL$DATABASE
-RDB-E-BAD_DB_FORMAT, SQL$DATABASE does not reference a database known to Rdb
-RMS-E-FNF, file not found
```

Example 2: Disconnecting a connection and starting a new connection with the same database

In this example, there are two connections: the default connection and a current connection, CA. Both connections use the personnel_ca database. Use the SHOW DATABASE statement to see the database settings.

```
SQL> --
SQL> -- Establish a default connection by attaching to the personnel_ca
SQL> -- database.
SQL> --
SQL> ATTACH 'FILENAME personnel_ca';
SQL> SHOW CONNECTIONS;
->      -default-
SQL> --
SQL> -- Start a new connection called CA.
SQL> --
SQL> CONNECT TO 'FILENAME personnel_ca'
cont>     AS 'CA';
SQL> SHOW CONNECTIONS;
        -default-
->      CA
SQL> --
SQL> -- The DISCONNECT CURRENT statement releases the connection name CA,
SQL> -- although the database personnel_ca still belongs to the default
SQL> -- connection.
SQL> --
SQL> DISCONNECT CURRENT;
SQL> SHOW CONNECTIONS;
->      -default-
SQL> --
SQL> -- Even though the database personnel_ca is still attached, CA
SQL> -- is no longer an active connection.
SQL> --
SQL> SET CONNECT 'CA';
%SQL-F-NOSUCHCON, There is not an active connection by that name
SQL> --
SQL> -- The original ATTACH statement comprises the default connection.
SQL> -- The DISCONNECT DEFAULT statement detaches the default connection.
SQL> --
SQL> DISCONNECT DEFAULT;
SQL> SHOW DATABASES;
%SQL-F-ERRATTDEF, Could not use database file specified by SQL$DATABASE
-RDB-E-BAD_DB_FORMAT, SQL$DATABASE does not reference a database known to Rdb
-RMS-E-FNF, file not found
```

## SET Control Statement

Assigns a value to a target parameter or a variable name.

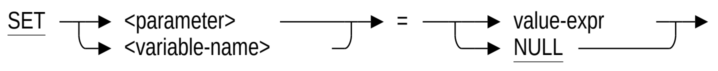
### Environment

You can use the SET assignment control statement in a compound statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

set-assignment-statement =



### Arguments

**parameter**
**variable-name**
Specifies the target where SQL stores a value expression or the NULL value.

**value-expr**
**NULL**
Assigns the value of a value expression or the NULL value to a target parameter or variable name.

### Usage Notes

- The data type of a value expression must be compatible with the data type of its target parameter or variable name.
- If you attempt to assign a value into a target specification that is shorter than the value, Oracle Rdb truncates the value and SQLSTATE returns a warning.

- If you do not use indicator parameters to identify NULL values and if the value expression is NULL, Oracle Rdb returns an error.

## Examples

Example 1: Assigning a value expression to a target parameter

```
BEGIN
SET :y = (SELECT COUNT (*) FROM employees);
END;
```

Example 2: Assigning the NULL value expression to a target parameter

```
BEGIN
SET :z = NULL;
END;
```

## SET DEFAULT CHARACTER SET Statement

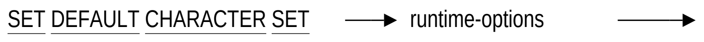
Specifies the default character set for the module or interactive SQL session.

### Environment

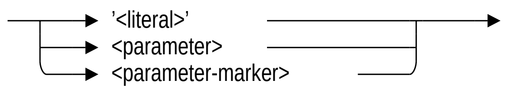
You can use the SET DEFAULT CHARACTER SET statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the default character set for your session. The value of runtime-options must be a valid character set. For a list of allowable character set names and option values, see Section 2.1.

### Usage Notes

- The SET DEFAULT CHARACTER SET statement sets the default character set for the session.

- If you have set the dialect to SQL92 or MIA, and if you do not specify the database default character set when you create the database, SQL assigns the session's default character set to the database default character set. Otherwise, SQL uses DEC_MCS as the default character set for the database.
- The session default character set may be set by issuing the DEFAULT CHARACTER SET clause within the SQL module header or by using the SET DEFAULT CHARACTER SET statement. See Section 2.1 for a list of default character sets.
- OpenVMS VAX OpenVMS Alpha If the session default character set was not specified within a module header or by using the SET DEFAULT CHARACTER SET statement and the logical RDB$CHARACTER_SET is defined, then SQL converts the value assigned to the logical name to a character set name. This character set is used as the module default character set. See Table E–2 for more information regarding conversion of logical names to character set names.

  The RDB$CHARACTER_SET logical name is deprecated and will not be supported in a future release. ♦
- Use the SHOW CHARACTER SET statement to display the current session character sets.

For information on setting the character sets for modules in SQL module language and precompiled SQL, see Section 3.2 and the DECLARE MODULE Statement.

## Example

Example 1: Setting the default character set of an interactive session

```
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
SQL>
SQL> SET DEFAULT CHARACTER SET 'DEC_KANJI';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_KANJI
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

## SET DEFAULT DATE FORMAT Statement

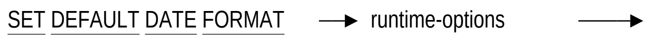
Specifies whether columns with the DATE data type or with the built-in function CURRENT_TIMESTAMP are interpreted as VMS or SQL92 format.

### Environment

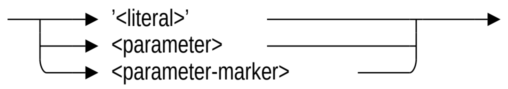
You can use the SET DEFAULT DATE FORMAT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the value of runtime-options, which must be one of the following:

- SQL92
- VMS

**SQL92**
Specifies that the DATE data type and the CURRENT_TIMESTAMP built-in function are interpreted as SQL92 format. The SQL92 format DATE contains only the YEAR TO DAY fields.

**VMS**
Specifies that the DATE data type and the CURRENT_TIMESTAMP built-in function are interpreted as VMS format. The VMS format DATE and CURRENT_TIMESTAMP contain YEAR TO SECOND fields.

The default is VMS.

## Usage Notes

- If the SET DIALECT statement is processed after the SET DEFAULT DATE FORMAT statement, it can override the setting of the SET DEFAULT DATE FORMAT statement.
- You cannot use the SET DEFAULT DATE FORMAT statement to modify the data type of a domain or column after it is created. Use the SET DEFAULT DATE FORMAT statement *before* you create a domain or column.
- Specifying the SET DEFAULT DATE FORMAT statement changes the default date format for the current connection only. Use the SHOW CONNECTIONS statement to display the characteristics of a connection.

## Example

Example 1: Setting CURRENT_TIMESTAMP to SQL92

In the following example, SQL issues an error message because, by default, CURRENT_TIMESTAMP is an OpenVMS data type and TIMESTAMP is in SQL92 format. The SET DEFAULT DATE FORMAT statement changes the default CURRENT_TIMESTAMP to SQL92 format.

```
SQL> SET DEFAULT DATE FORMAT 'VMS';
SQL> --
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP
cont>              DEFAULT CURRENT_TIMESTAMP;
%SQL-F-DEFVALINC, You specified a default value for LOGGING_DATE which is
inconsistent with its data type
SQL> SET DEFAULT DATE FORMAT 'SQL92';
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP
cont>              DEFAULT CURRENT_TIMESTAMP;
SQL> SHOW DOMAINS LOGGING_DATE;
LOGGING_DATE                   TIMESTAMP(2)
 Rdb default: CURRENT_TIMESTAMP
```

## SET DIALECT Statement

Specifies the settings of the current connection for the following characteristics:

- Whether the length of character string parameters, columns, and domains are interpreted as characters or octets. This can also be specified by using the SET CHARACTER LENGTH statement.
- Whether double quotation marks are interpreted as string literals or delimited identifiers. This can also be specified by using the SET QUOTING RULES statement.
- Whether or not identifiers can be keywords. This can also be specified by using the SET KEYWORD RULES statement.
- Which views are read-only. This can also be specified by using the SET VIEW UPDATE RULES statement.
- Whether columns with the DATE or CURRENT_TIMESTAMP data type are interpreted as VMS or SQL92 format. This can also be specified by using the SET DEFAULT DATE FORMAT statement.
- Whether character sets change. Character sets can be changed using the SET DEFAULT CHARACTER SET, SET NATIONAL CHARACTER SET, SET IDENTIFIER CHARACTER SET, and SET LITERAL CHARACTER SET statements.

The SET DIALECT statement lets you specify several settings with one command, instead of specifying each setting individually.

Table 7–5 shows the settings for each option.

**Table 7–5 Dialect Settings**

| Characteristic | SQL92 | SQL89 | MIA | SQLV40 | ORACLE LEVEL1 |
|---|---|---|---|---|---|
| Character length | Characters | Characters | Characters | Octets | Characters |
| Quoting rules | Delimited identifier | Delimited identifier | Delimited identifier | Literal | Delimited identifier |
| Keywords allowed as identifiers | No | No | No | Yes | Yes |
| View update rules | ANSI/ISO SQL rules | ANSI/ISO SQL rules | ANSI/ISO SQL rules | Oracle Rdb rules | ANSI/ISO SQL rules |
| Default date format | DATE ANSI | DATE ANSI | DATE ANSI | DATE VMS | DATE VMS |
| Parameters | Required | Required | Required | Not allowed | Not applicable |
| Default character set | Not changed | Not changed | KATAKANA | Not changed | Not changed |
| National character set | Not changed | Not changed | KANJI | Not changed | Not changed |
| Identifier character set | Not changed | Not changed | DEC_KANJI | Not changed | Not changed |
| Literal character set | Not changed | Not changed | KATAKANA | Not changed | Not changed |

Oracle Rdb recommends that you set the dialect to SQL92 or MIA, unless you need to maintain compatibility with an earlier dialect.

## Environment

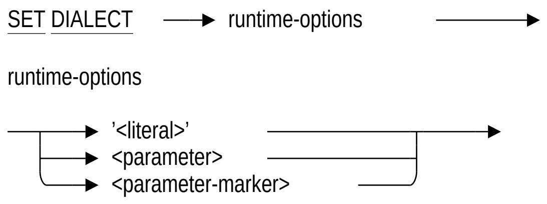
You can use the SET DIALECT statement:

- In interactive SQL
- Embedded in host language programs to be precompiled to effect the processing of dynamic SQL statements (use the DIALECT clause to effect dialect changes in the precompiled source)
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

However, the ORACLE LEVEL1 dialect can be used only in the interactive SQL and dynamic SQL environments.

## SET DIALECT Statement

### Format



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the value of the runtime-options, which must be one of the following:

- SQL92
- SQL89
- MIA
- SQLV40
- ORACLE LEVEL1

**SQL92**
Specifies the following behavior:

- The length of character string parameters, columns, and domains is interpreted as characters, rather than octets.
- Double quotation marks are interpreted as delimited identifiers.
- Keywords cannot be used as identifiers unless they are enclosed within double quotation marks.
- The ANSI/ISO SQL standard for updatable views is applied to all views created during compilation. Views that do not comply with the ANSI/ISO SQL standard for updatable views cannot be updated.

  The ANSI/ISO SQL standard for updatable views requires the following conditions to be met in the SELECT statement:

  – The DISTINCT keyword is not specified.

  - Only column names can appear in the select list. Each column name can appear only once. Functions and expressions such as max(column_name) or column_name +1 cannot appear in the select list.
  - The FROM clause refers to only one table. This table must be either a base table, global temporary table, local temporary table, or a derived table that can be updated.
  - The WHERE clause does not contain a subquery.
  - The GROUP BY clause is not specified.
  - The HAVING clause is not specified.
- The DATE and CURRENT_TIMESTAMP data types are interpreted as SQL92 format. The SQL92 format DATE contains only the YEAR TO DAY fields.
- Conversions between character data types when storing data or retrieving data raise exceptions or warnings in certain situations. For further explanation of these situations, see Section 2.3.7.2.
- You can specify DECIMAL or NUMERIC for formal parameters in SQL modules and declare host language parameters with packed decimal or signed numeric storage format. SQL generates an error message if you attempt to exceed the precision specified.
- The USER keyword specifies the current active user name for a request.
- A warning is generated when a NULL value is eliminated from a SET function.
- The WITH CHECK OPTION clause on views returns a discrete error code from an integrity constraint failure.
- An exception is generated with terminated C strings that are not NULL.

If you set the dialect to SQL92, the default on constraint evaluation time is set to NOT DEFERRABLE.

**SQL89**
**MIA**
Specifies the following behavior:

- The length of character string parameters, columns, and domains is interpreted as characters, rather than octets.
- Double quotation marks are interpreted as delimited identifiers.
- Keywords cannot be used as identifiers unless they are enclosed within double quotation marks.

- The ANSI/ISO SQL standard for updatable views is applied to all views created during compilation. Views that do not comply with the ANSI/ISO SQL standard for updatable views cannot be updated.

  The ANSI/ISO SQL standard for updatable views requires the following conditions to be met in the SELECT statement:

  - The DISTINCT keyword is not specified.
  - Only column names can appear in the select list. Each column name can appear only once. Functions and expressions such as max(column_name) or column_name +1 cannot appear in the select list.
  - The FROM clause refers to only one table. This table must be either a base table, global temporary table, local temporary table, or a derived table that can be updated.
  - The WHERE clause does not contain a subquery.
  - The GROUP BY clause is not specified.
  - The HAVING clause is not specified.

If you specify MIA, SQL sets the character sets as follows:

- Default character set: KATAKANA
- National character set: KANJI
- Identifier character set: DEC_KANJI
- Literal character set: KATAKANA

If you set the dialect to SQL89 or MIA, the constraint evaluation time is DEFERRABLE.

**SQLV40**
Specifies the following behavior:

- The length of character string parameters, columns, and domains is interpreted as octets, rather than characters.
- Double quotation marks are interpreted as string literals.
- Keywords can be used as identifiers.
- The ANSI/ISO SQL standard for updatable views is not applied. Instead, SQL considers views that meet the following conditions to be updatable:
  - The DISTINCT keyword is not specified.

  - The FROM clause refers to only one table. This table must be either a base table, global temporary table, local temporary table, or a derived table that can be updated.
  - The WHERE clause does not contain a subquery.
  - The GROUP BY clause is not specified.
  - The HAVING clause is not specified.
- The DATE and CURRENT_TIMESTAMP data types are interpreted as VMS format. The VMS format DATE and CURRENT_TIMESTAMP contain YEAR TO SECOND fields.

If you set the dialect to SQLV40, the constraint evaluation time is DEFERRABLE.

The default is SQLV40.

See Table 7–5 for the setting values of the dialect options.

**ORACLE LEVEL1**
Specifies the following behavior:

- The same dialect rules as SQL92 are in effect minus reserved word checking and the DATE ANSI format.
- The ORACLE LEVEL1 dialect allows the use of aliases to reference (or link) to tables in data manipulation statements like SELECT, DELETE, INSERT, and UPDATE. For example:

```
SQL> ATTACH 'ALIAS pers_alias FILENAME mf_personnel';
SQL> SET DIALECT 'ORACLE LEVEL1';
SQL> SELECT * FROM employees@pers_alias
cont> WHERE employee_id = '00164';
 EMPLOYEE_ID   LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
   ADDRESS_DATA_1              ADDRESS_DATA_2         CITY
      STATE   POSTAL_CODE   SEX    BIRTHDAY      STATUS_CODE
 00164         Toliver          Alvin        A
   146 Parnell Place                                  Chocorua
      NH      03817         M      28-Mar-1947   1

1 row selected
```

  Alias references are only allowed on the table name and not on column names. You cannot put a space between the table name, the at (@) sign, and the alias name.

  If you specify a schema name when referencing an Oracle Rdb database, the schema name is ignored unless the multischema attribute is on.

- Comments embedded in SQL queries are allowed in the following format:

  ```
  SQL> SELECT * FROM
  cont>               /* this is a comment */
  cont> EMPLOYEES LIMIT TO 1 ROW;
   EMPLOYEE_ID   LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
     ADDRESS_DATA_1              ADDRESS_DATA_2         CITY
        STATE   POSTAL_CODE   SEX    BIRTHDAY      STATUS_CODE
   00165        Smith            Terry        D
     120 Tenby Dr.                                   Chocorua
        NH      03817         M      15-May-1954   2

  1 row selected
  ```

- The following basic predicates for inequality comparisons are supported:

  ```
  <>
  ^=
  !=
  ```

  The <> and ^= basic predicates are also supported in other dialects. The != basic predicate requires that the ORACLE LEVEL1 dialect be set to avoid confusion with the interactive SQL comment character.

- When using dynamic SQL, the client application can specify a synonym for the parameter marker (?). For example, :name, :l, :1, :2, and so on.
- In a SELECT statement, the FOR UPDATE OF clause provides UPDATE ONLY CURSOR semantics by locking all the rows selected.
- The string concatenation operator and the CONCAT function treat nulls as zero-length strings.
- The default date format is DATE VMS which is capable of doing arithmetic in the ORACLE LEVEL1 dialect only. Addition and subtraction can be done with numeric data types that are implicitly cast to the INTERVAL DAY data type. Fractions are rounded to the nearest whole integer.
- Zero length strings are null. When using an Oracle7 database, a VARCHAR of zero length is considered null. While the Oracle Rdb ORACLE LEVEL1 dialect does not remove zero length strings from the database, it does make them difficult to create. The following rules are in effect:
  - Empty literal strings (for example, `''`) are considered literal nulls.
  - Any function that encounters a zero length string returns a null in its place. This includes stored and external functions returning a VARCHAR data type regardless of the dialect under which they were compiled. It also includes the TRIM and SUBSTRING built-in functions.

– Parameters with the VARCHAR data type and a length of zero are treated as null.

The best way to avoid zero length strings from being seen by an Oracle7 application is to only use views compiled under the ORACLE LEVEL1 dialect and to modify tables with VARCHAR columns to remove zero length strings. The following example shows how to removes zero length strings from a VARCHAR column in a table:

```
SQL> UPDATE tab1 SET col1 = NULL WHERE CHARACTER_LENGTH(col1) = 0;
```

If modifying the table is not possible or if a view compiled in another dialect containing VARCHAR functions must be used, then create a new view under the ORACLE LEVEL1 dialect referring to that table or view to avoid the zero length VARCHAR string. The following example shows how to avoid selecting zero length strings from a VARCHAR column in a table or non-Oracle dialect view:

```
SQL> SET DIALECT 'ORACLE LEVEL1';
SQL> CREATE VIEW view1 (col1, col2)
cont>    AS SELECT SUBSTRING(col1 FROM 1 FOR 2000), col2 FROM tab1;
```

The Oracle Rdb optimizer is more efficient if data is selected without the use of functions. Therefore, the previous example is best used only if you suspect zero length strings have been inserted into the table and it is necessary to avoid them.

- The SQLCA SQLERRD[0] field returns a nonzero value.
- The ROWNUM keyword is allowed in select expressions and limits the number of rows returned in the query. The following example limits the number of rows returned by the SELECT statement to 9 rows:

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> SET DIALECT 'ORACLE LEVEL1';
SQL> SELECT last_name FROM EMPLOYEES WHERE ROWNUM < 10;
 LAST_NAME
 Toliver
 Smith
 Dietrich
 Kilpatrick
 Nash
 Gray
 Wood
 D'Amico
 Peters
9 rows selected
```

Conditions testing for ROWNUM values greater than or equal to a positive integer are always false and, therefore, return no rows. For example:

```
SQL> SELECT last_name FROM EMPLOYEES WHERE ROWNUM > 10;
0 rows selected
SQL> SELECT last_name FROM EMPLOYEES WHERE ROWNUM = 10;
0 rows selected
```

See the Usage Notes for additional restrictions that apply to the ROWNUM keyword.

- The NEXTVAL and CURRVAL column names are reserved for future use.

## Usage Notes

- If the following statements are processed after the SET DIALECT statement, they override the settings of the SET DIALECT statement:
  - SET CHARACTER LENGTH
  - SET QUOTING RULES
  - SET KEYWORD RULES
  - SET VIEW UPDATE RULES
  - SET DEFAULT DATE FORMAT
  - SET DEFAULT CHARACTER SET
  - SET NATIONAL CHARACTER SET
  - SET IDENTIFIER CHARACTER SET
  - SET LITERAL CHARACTER SET
  - SET NAMES

  These statements change the settings of the current connection only.
- If you specify MIA and then change the dialect to another value, the MIA character sets remain intact for the default, national, identifier, and literal character sets. You must manually change the character set for each of these in this situation. For more information on changing the session character sets, see the SET DEFAULT CHARACTER SET Statement, the SET IDENTIFIER CHARACTER SET Statement, the SET LITERAL CHARACTER SET Statement, and the SET NATIONAL CHARACTER SET Statement.

- Use the SHOW CONNECTIONS statement to display the characteristics of a connection.
- If the source string is greater than the target string when converting between character data types, the result is left-justified and truncated on the right with no error reported for dialects MIA, SQL89, and SQLV40.

  If you have set your dialect to SQL92, an error is returned when storing data unless the truncated characters are only space characters in which case no error is returned. If you are retrieving data, a warning is returned if truncation occurs. The warning is returned regardless of whether or not the truncated characters are blank.
- If you set your dialect to SQL89, Oracle Rdb allows the translation of a missing value (defined using the RDO interface) to process when inserting or updating data in the database using the SQL interface. If a value is set to the missing value using RDO, the resulting value of an insert or update using SQL is NULL.
- If you have used ROWNUM as a column name, consider changing the column name. Otherwise, when referencing the column, you need to either use single quotation marks around the column name or precede it with the table name.
- Other restrictions that apply to the ROWNUM keyword are:
  – Can be used only with the ORACLE LEVEL1 dialect. All other dialects must use the LIMIT TO clause.
  – Can be used only in a comparison of select expression predicate.
  – Can appear only in SELECT statements or select expressions.
  – Cannot be used with a LIMIT TO clause.
  – Cannot appear more than once in the predicate of a WHERE clause.
  – Cannot be compared to a column.
  – Cannot be used in a compound statement.
  – Cannot appear on either side of an OR Boolean operator.
  – Cannot be selected or used in a function call.

## Example

Example 1: Setting the characteristics to SQL92

```
SQL> ATTACH 'ALIAS MIA1 FILENAME MIA_CHAR_SET';
SQL> CONNECT TO 'ALIAS MIA1 FILENAME MIA_CHAR_SET' AS 'TEST';
SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: SQLV40
Default DATE type: DATE VMS
Quoting Rules: SQLV40
Optimization Level: DEFAULT
Hold Cursors default: WITH HOLD PRESERVE NONE
Quiet commit mode: OFF
Compound transactions mode: EXTERNAL
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
SQL> --
SQL> -- Change the environment from SQLV40 to MIA.  Notice that the session
SQL> -- character sets change.
SQL> --
SQL> SET DIALECT 'MIA';
SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: MIA
Default character unit: CHARACTERS
Keyword Rules: MIA
View Rules: ANSI/ISO
Default DATE type: DATE ANSI
Quoting Rules: ANSI/ISO
Optimization Level: DEFAULT
Hold Cursors default: WITH HOLD PRESERVE NONE
Quiet commit mode: OFF
Compound transactions mode: EXTERNAL
Default character set is KATAKANA
National character set is KANJI
```

```
Identifier character set is DEC_KANJI
Literal character set is KATAKANA

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
SQL> --
SQL> -- Change the environment from MIA to SQL92.  Notice that the
SQL> -- session characters DO NOT change from the MIA settings.
SQL> --
SQL> SET DIALECT 'SQL92';
SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQL92
Default character unit: CHARACTERS
Keyword Rules: SQL92
View Rules: ANSI/ISO
Default DATE type: DATE ANSI
Quoting Rules: ANSI/ISO
Optimization Level: DEFAULT
Hold Cursors default: WITH HOLD PRESERVE NONE
Quiet commit mode: ON
Compound transactions mode: EXTERNAL
Default character set is KATAKANA
National character set is KANJI
Identifier character set is DEC_KANJI
Literal character set is KATAKANA

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
```

## SET FLAGS Statement

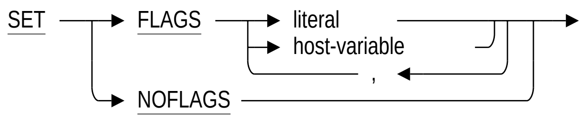
Allows enabling and disabling of database system debug flags for the current session.

### Environment

You can use the SET FLAGS statement:

- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**FLAGS**
Specifies whether or not a database system debug flag is set.

Table 7–6 shows the available keywords that can be specified.

**Table 7–6 Debug Flag Keywords**

| Keyword | Negated Keyword | Debug Flags Equivalent[1] | Comment |
|---|---|---|---|
| BLR | NOBLR | Bcn[2] | Displays the binary language representation request for the query |
| EXECUTION | NOEXECUTION | E[2] | Displays an execution trace from the dynamic optimizer |

[1]RDMS$DEBUG_FLAGS logical name on OpenVMS and RDB_DEBUG_FLAGS configuration parameter on Digital UNIX

[2]Equivalent to the keyword listed in the Keyword column

(continued on next page)

**Table 7–6 (Cont.) Debug Flag Keywords**

| Keyword | Negated Keyword | Debug Flags Equivalent[1] | Comment |
|---|---|---|---|
| ITEM_LIST | NOITEM_LIST | H[2] | Displays item list information passed in for the database queries and as compile-time query options |
| CARDINALITY | NOCARDINALITY | K[2] | Shows cardinality updates |
| MBLR | NOMBLR | M[2] | Displays the metadata binary language representation request for the data definition language statement |
| ESTIMATES | NOESTIMATES | O[2] | Displays the optimizer estimates |
| DATABASE_ PARAMETERS | NODATABASE_ PARAMETERS | P[2] | Displays the database parameter buffer during ATTACH, CREATE, ALTER, and DISCONNECT |
| SORT_STATISTICS[5] | NOSORT_STATISTICS[5] | R[2] | Shows sort statistics during execution |
| STRATEGY | NOSTRATEGY | S[2] | Shows the optimizer strategy |
| TRANSACTION_ PARAMETERS | NOTRANSACTION_ PARAMETERS | T[2] | Displays the transaction parameter buffer during SET TRANSACTION, COMMIT, and ROLLBACK and during stored procedure compilation |
| CONTROL_BITS | NOCONTROL_BITS | Bc[2] | Used with BLR keyword to display a decoding of the BLR$K_CONTROL_BITS semantic flags |

[1]RDMS$DEBUG_FLAGS logical name on OpenVMS and RDB_DEBUG_FLAGS configuration parameter on Digital UNIX

[2]Equivalent to the keyword listed in the Keyword column

[5]Available on the OpenVMS platform only.

(continued on next page)

**Table 7–6 (Cont.) Debug Flag Keywords**

| Keyword | Negated Keyword | Debug Flags Equivalent[1] | Comment |
|---|---|---|---|
| PREFIX[4] | NOPREFIX | Bn[3] | Used with BLR keyword to inhibit offset numbering and other formatting of binary language representation display |
| CRONO_FLAG | NOCRONO_FLAG | Xc[2] | Forces timestamp-before-dump display |
| OUTLINE | NOOUTLINE | Ss[2] | Displays query outline for this query (can be used without STRATEGY keyword) |
| REQUEST_NAMES | NOREQUEST_ NAMES | Sn[2] | Displays the names of user requests, triggers, and constraints |
| TRACE | NOTRACE | Xt[2] | Enables output from TRACE statement |
| SCROLL_EMULATION | NOSCROLL_ EMULATION | L[2] | Disables scrolling for old-style LIST OF BYTE VARYING (segmented string) format, which is either the RDMS$DIAG_ FLAGS logical name on OpenVMS or RDB_ DIAG_FLAGS configuration parameter on Digital UNIX |

[1]RDMS$DEBUG_FLAGS logical name on OpenVMS and RDB_DEBUG_FLAGS configuration parameter on Digital UNIX

[2]Equivalent to the keyword listed in the Keyword column

[3]Equivalent to the keyword listed in the Negated Keyword column

[4]Enabled by default

(continued on next page)

**Table 7–6 (Cont.) Debug Flag Keywords**

| Keyword | Negated Keyword | Debug Flags Equivalent[1] | Comment |
|---|---|---|---|
| SORTKEY_EXT | NOSORTKEY_EXT | S[2] | Reports if ORDER BY (or SORTED BY) is referencing only external (constant) value, which is either the RDMS$DIAG_FLAGS logical name on OpenVMS or the RDB_DIAG_FLAGS configuration parameter on Digital UNIX |
| IGNORE_OUTLINE | NOIGNORE_ OUTLINE | I[2] | Ignores outlines defined in the database, which is an RDMS$BIND_OUTLINE_ FLAGS logical name on OpenVMS or the RDB_ BIND_OUTLINE_FLAGS configuration parameter on Digital UNIX |

[1]RDMS$DEBUG_FLAGS logical name on OpenVMS and RDB_DEBUG_FLAGS configuration parameter on Digital UNIX

[2]Equivalent to the keyword listed in the Keyword column

**NOFLAGS**
Disables all previously set flags for the current session until the SET FLAGS statement is specified.

## Usage Notes

- The specified flag is processed by each base database to which you are currently attached.
- The SET FLAGS statements overrides the RDMS$DEBUG_FLAGS logical name or the RDB_DEBUG_FLAGS configuration parameter set at the command level.
- The TRACE flag can be used from any stored routine. However, because stored routines (nested or otherwise) are only loaded once per session, the TRACE flag must be enabled before invoking the routines.

- The Oracle Rdb optimizer may evaluate SQL functions in a different order than was specified. Thus the order (or frequency) of the TRACE display may not be predictable.
- The keywords can be abbreviated to the smallest nonambiguous length. The minimum length is 2 characters.
- Upper- and lowercase are equivalent for keywords.
- The SET FLAGS statement must be executed after a database attach has been established, otherwise the statement has no effect on the database.
- The SET FLAGS statement does not persist beyond a database attach.
- The EXECUTION keyword can be followed immediately by a numeric value in parentheses. This represents the number of lines to display for stopping the execution trace for query execution. The default is 100. For example:

  ```
  SQL> SET FLAGS 'EXECUTION(1000)';
  SQL> SHOW FLAGS

  Alias RDB$DBHANDLE:
  Flags currently set for Oracle Rdb:
     PREFIX,EXECUTION(1000)
  ```

  There cannot be a space between the keyword and the numeric value in parentheses.

## Examples

Example 1: Enabling and disabling database system debug flags

```
SQL> ATTACH 'FILENAME MF_PERSONNEL';
SQL> SHOW FLAGS

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX
SQL>
SQL> SET FLAGS 'TRACE';
SQL> SHOW FLAGS

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,TRACE
SQL>
SQL> SET FLAGS 'STRATEGY';
SQL> SHOW FLAGS
```

```
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   STRATEGY,PREFIX,TRACE
SQL>
SQL> SET FLAGS 'NOTRACE';
SQL> SHOW FLAGS

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   STRATEGY,PREFIX
SQL>
SQL> SET NOFLAGS;
SQL> SHOW FLAGS

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX
SQL>
```

Example 2: Using the PREFIX keyword

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> -- Show that the PREFIX keyword is enabled by default
SQL> --
SQL> SHOW FLAGS

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX
SQL> --
SQL> -- Enable TRACE
SQL> --
SQL> SET FLAGS 'TRACE';
SQL> SHOW FLAGS

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,TRACE
SQL> --
SQL> -- Show that the prefix is displayed
SQL> --
SQL> BEGIN
cont>  TRACE 'AAA';
cont> END;
~Xt: AAA
SQL> --
SQL> -- Turn off the prefix
SQL> --
SQL> SET FLAGS 'NOPREFIX';
SQL> SHOW FLAGS
```

```
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   TRACE
SQL> --
SQL> -- Show that the prefix is no longer displayed
SQL> --
SQL> BEGIN
cont>  TRACE 'AAA';
cont> END;
AAA
```

## SET HOLD CURSORS Statement

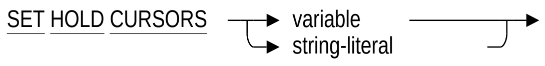
Specifies the session default attributes for holdable cursors that have not been previously defined.

### Environment

You can use the SET HOLD CURSORS statement:

- In interactive SQL
- Embedded in host language programs to be precompiled to change the behavior of dynamic cursors
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**variable**
**string-literal**
Specifies the attribute for the holdable cursor. Values can include:

- ON COMMIT

  All cursors declared without a WITH HOLD clause or with a WITH HOLD ON COMMIT clause remain open when you commit.
- ON ROLLBACK

  All cursors declared without a WITH HOLD clause or with a WITH HOLD ON ROLLBACK clause remain open when you roll back.
- ALL

  All cursors remain open with the exception of those declared with a WITH HOLD clause.

- NONE

  All cursors close with the exception of those declared with a WITH HOLD clause.

  This is the default if you do not specify a SET HOLD CURSORS statement.

## Usage Notes

- Cursors defined prior to the SET HOLD CURSORS statement are not affected.
- ODBC driver requires the holdable cursor be set as a connection attribute.
- The string-literal must be inside single quotation marks (').

## Example

Example 1: Setting session default attributes for holdable cursors

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> -- Define the session default
SQL> --
SQL> SET HOLD CURSORS 'ON ROLLBACK';
SQL> --
SQL> -- Declare the cursor
SQL> --
SQL> DECLARE curs1 CURSOR FOR
cont> SELECT first_name, last_name FROM employees;
SQL> OPEN curs1;
SQL> FETCH curs1;
 FIRST_NAME   LAST_NAME
 Terry        Smith
SQL> FETCH curs1;
 FIRST_NAME   LAST_NAME
 Rick         O'Sullivan
SQL> DELETE FROM employees WHERE CURRENT OF curs1;
1 row deleted
SQL> ROLLBACK;
SQL> FETCH curs1;
 FIRST_NAME   LAST_NAME
 Stan         Lasch
SQL> COMMIT;
SQL> FETCH curs1;
%SQL-F-CURNOTOPE, Cursor CURS1 is not opened
```

Example 2: Overriding the session default attributes for holdable cursors

```
SQL> -- Set the session default
SQL> --
SQL> SET HOLD CURSORS 'ALL';
SQL> --
SQL> -- Declare the cursor without a WITH HOLD clause
SQL> --
SQL> DECLARE curs2 CURSOR FOR
cont> SELECT first_name, last_name FROM employees;
SQL> OPEN curs2;
SQL> FETCH curs2;
 FIRST_NAME   LAST_NAME
 Terry        Smith
SQL> FETCH curs2;
 FIRST_NAME   LAST_NAME
 Rick         O'Sullivan
SQL> ROLLBACK;
SQL> FETCH curs2;
 FIRST_NAME   LAST_NAME
 Stan         Lasch
SQL> COMMIT;
SQL> FETCH curs2;
 FIRST_NAME   LAST_NAME
 Susan        Gray
SQL> CLOSE curs2;
SQL> FETCH curs2;
%SQL-F-CURNOTOPE, Cursor CURS2 is not opened
SQL> --
SQL> -- Declare the cursor overriding the session default by
SQL> -- specifying the WITH HOLD clause
SQL> --
SQL> DECLARE curs3 CURSOR
cont> WITH HOLD PRESERVE ON COMMIT
cont> FOR SELECT first_name, last_name FROM employees;
SQL> OPEN curs3;
SQL> FETCH curs3;
 FIRST_NAME   LAST_NAME
 Terry        Smith
SQL> FETCH curs3;
 FIRST_NAME   LAST_NAME
 Rick         O'Sullivan
SQL> COMMIT;
SQL> FETCH curs3;
 FIRST_NAME   LAST_NAME
 Stan         Lasch
SQL> ROLLBACK;
SQL> FETCH curs3;
%SQL-F-CURNOTOPE, Cursor CURS3 is not opened
```

## SET IDENTIFIER CHARACTER SET Statement

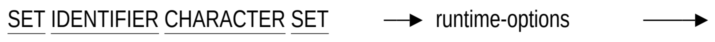
Specifies the identifier character set for the module or interactive SQL session.

### Environment

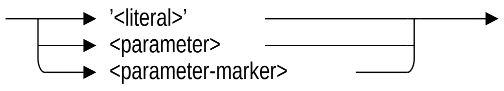
You can use the SET IDENTIFIER CHARACTER SET statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the character set used for database object names such as table names and column names. The value of runtime-options must be a valid character set. See Table 2-3 for a list of allowable character sets and option values.

### Usage Notes

- The SET IDENTIFIER CHARACTER SET statement sets the identifier character set for the session.
- The specified identifier character set must contain ASCII characters. See Table 2-3 for a list of allowable character sets.

- If you set the dialect to SQL92 or MIA, and if you do not specify the identifier character set when you create the database, SQL uses the session's identifier character set. Otherwise, SQL uses DEC_MCS as the identifier character set for the database.
- The identifier character set of the session should match the identifier character set of all attached databases.
- The identifier character set also specifies the character set for the SQLNAME field in SQLDA and SQLDA2 for statements without an explicit database context.
- Use the SHOW CHARACTER SETS statement to display the current session character sets.

For information on setting the character sets for modules in SQL module language and precompiled SQL, see Section 3.2 and the DECLARE MODULE Statement.

## Example

Example 1: Setting the identifier character set of an interactive session

```
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
SQL>
SQL> SET IDENTIFIER CHARACTER SET 'DEC_KANJI';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_KANJI
Literal character set is DEC_MCS
```

## SET KEYWORD RULES Statement

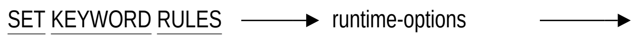
Specifies whether or not you can use identifiers as keywords in the current attach.

### Environment

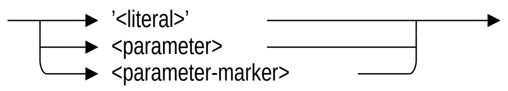
You can use the SET KEYWORD RULES statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**′literal′**
**parameter**
**parameter-marker**
Specifies the value of runtime-options, which must be one of the following:

- SQL92
- SQL89
- MIA
- SQLV40

**SQL92**
**SQL89**
**MIA**
Specifies that SQL returns an error if statements use keywords (reserved words) as identifiers, unless the keywords are enclosed in double quotation marks.

**SQLV40**
Specifies that SQL allows statements to use keywords as identifiers. SQL issues an informational message that the use of keywords as identifiers is a deprecated feature. See Appendix F for more information regarding deprecated features.

The default is SQLV40.

## Usage Notes

- If the SET DIALECT statement is processed after the SET KEYWORD RULES statement, it overrides the setting of the SET KEYWORD RULES statement.
- The SET KEYWORD RULES statement implicitly sets the quoting rules. If the SET QUOTING RULES statement is processed after the SET KEYWORD RULES statement, it overrides the quoting rules implicitly set by the SET KEYWORD RULES statement.
- If the SET KEYWORD RULES statement is processed after the SET QUOTING RULES statement, it overrides the quoting rules set by the SET QUOTING RULES statement.
- Specifying the SET KEYWORD RULES statement changes the keyword and quoting rules for the current attach only. Use the SHOW CONNECTIONS statement to display the characteristics of an attach.

## Examples

Example 1: Setting the keyword rule characteristics to SQL92

```
SQL> SET KEYWORD RULES ’SQL92’;
SQL> --
SQL> -- Because NATIONAL is a keyword, SQL returns an error message.
SQL> --
SQL> CREATE DOMAIN NATIONAL CHAR (2);
%SQL-F-RES_WORD_AS_IDE, Keyword NATIONAL used as an identifier
SQL> --
SQL> -- Enclose NATIONAL in double quotation marks.
SQL> --
SQL> CREATE DOMAIN "NATIONAL" CHAR (2);
SQL> --
```

Example 2: Setting the keyword rule characteristics to SQLV40

```
SQL> SET KEYWORD RULES ’SQLV40’;
SQL> --
SQL> -- You can use a keyword as an identifier.
SQL> --
SQL> CREATE DOMAIN NATIONAL CHAR (2);
%SQL-I-DEPR_FEATURE, Deprecated Feature: Keyword national used as an identifier
SQL> --
```

## SET LITERAL CHARACTER SET Statement

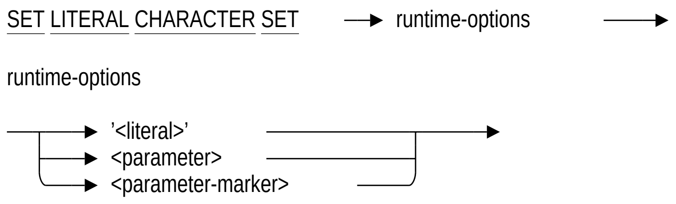
Specifies the literal character set for the module or interactive SQL session.

### Environment

You can use the SET LITERAL CHARACTER SET statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the character set for literals that are not qualified by a character set or national character set. The value of runtime-options must be a valid character set. See Section 2.1 for a list of the allowable character sets and option values.

### Usage Notes

- The SET LITERAL CHARACTER SET statement sets the literal character set for the session.

- If you set the dialect to MIA, the literal character set is KATAKANA. Otherwise, if you do not set a dialect or change the literal character set, SQL uses DEC_MCS.
- Use the SHOW CHARACTER SETS statement to display the current session character sets.

## Example

Example 1: Setting the literal character set of an interactive session

```
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
SQL> --
SQL> SET LITERAL CHARACTER SET 'DEC_KANJI';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_KANJI
```

## SET NAMES Statement

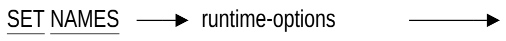
Specifies the default, identifier, and literal character sets for the session. The SET NAMES statement also specifies the character parameters for SQL module language.

### Environment

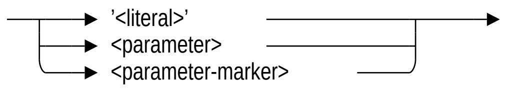
You can use the SET NAMES statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**′literal′**
**parameter**
**parameter-marker**
Specifies the character set used for the default, identifier, and literal character set for the session. The value of runtime-options must be a valid character set. See Table 2-3 for a list of allowable character sets and option values.

## Usage Notes

- The SET NAMES statement sets the identifier, default, and literal character sets for the session and overrides any previous changes. If you want the identifier, default, or literal character set to be different than the character set specified in the SET NAMES statement, specify it after issuing the SET NAMES statement.
- The specified character set must contain ASCII characters. See Table 2-3 for a list of allowable character sets.
- The SET NAMES statement also specifies the character set for the SQLNAME field in SQLDA and SQLDA2 for statements without an explicit database context.
- Use the SHOW CHARACTER SETS statement to display the current session character sets.

For information on setting the character sets for modules in SQL module language and precompiled SQL, see Section 3.2 and the DECLARE MODULE Statement.

## Example

Example 1: Setting the default, identifier, and literal character sets of an interactive session

```
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Names character set is DEC_MCS
Literal character set is DEC_MCS
SQL> --
SQL> SET NAMES 'DEC_KANJI';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_KANJI
National character set is DEC_MCS
Names character set is DEC_KANJI
Literal character set is DEC_KANJI
SQL> --
SQL> -- Specifying a different default character set
SQL> --
SQL> SET DEFAULT CHARACTER SET 'DEC_KOREAN';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_KOREAN
National character set is DEC_MCS
Names character set is DEC_KANJI
Literal character set is DEC_KANJI
```

## SET NATIONAL CHARACTER SET Statement

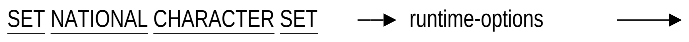
Specifies the national character set for the module or interactive SQL session.

### Environment

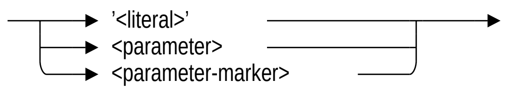
You can use the SET NATIONAL CHARACTER SET statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the national character set for your session. The value of runtime-options must be a valid character set. For a list of allowable character set names and option values, see Section 2.1.

### Usage Notes

- The SET NATIONAL CHARACTER SET statement sets the national character set for the session.

- The national character set determines the character set for character string literals qualified by the national character set, NCHAR, and NCHAR VARYING. Section 2.1 lists the character sets you can use for the national character set for the database.
- If you have set the dialect to SQL92 or MIA, and if you do not specify the national character set when you create the database, SQL uses the session's national character set. Otherwise, SQL uses DEC_MCS as the national character set.
- Use the SHOW CHARACTER SETS statement to display the current session character sets.

For information on setting the character sets for modules in SQL module language and precompiled SQL, see Section 3.2 and the DECLARE MODULE Statement.

## Example

Example 1: Setting the national character set for an interactive session

```
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
SQL>
SQL> SET NATIONAL CHARACTER SET 'KANJI';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is KANJI
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

## SET OPTIMIZATION LEVEL Statement

Requests either the fast-first record retrieval strategy or total-time retrieval strategy of optimization for any given query. You can also set or reset the defaults for either the fast-first record retrieval or total-time retrieval strategies. The SET OPTIMIZATION LEVEL statement specifies the query optimization level for dynamic SQL query compilation only; the statement does not affect the SQL compile-time environment nor does it affect the run-time environment of static SQL queries.

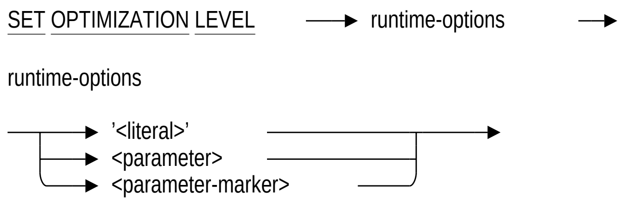
See Chapter 3 and Chapter 4 for information on setting the optimization level in SQL module and precompiler languages.

### Environment

You can use the SET OPTIMIZATION LEVEL statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the value of the runtime-options, which must be one of the following:

- FAST FIRST
- TOTAL TIME

- DEFAULT

**FAST FIRST**
If the majority of the queries in your session are going to be terminated before all the result records are delivered, use the FAST FIRST option.

If optimization strategy is not explicitly set, FAST FIRST is the default.

**TOTAL TIME**
If the majority of the queries in your session are not going to be closed before all the result records are delivered, use the TOTAL TIME option.

Most queries benefit from TOTAL TIME optimization.

**DEFAULT**
Sets the optimization back to the default behavior which is to try the fast-first retrieval strategy first, then select the total-time retrieval strategy if it will retrieve the records faster than the fast-first strategy.

## Usage Notes

- You can set the most commonly used optimization level in your initialization procedure (the SQLINI.SQL procedure that is automatically executed in the beginning of each session).
- You can change the optimization level default for a particular query (not just for cursors as with previous versions of Oracle Rdb) by specifying a SELECT statement. See the SELECT Statement: General Form for more detail on setting the optimization level with a SELECT statement.

## Example

Example 1: Setting the optimization level

You must define the logical name RDMS$DEBUG_FLAGS or configuration parameter RDB_DEBUG_FLAGS to be "S" before you execute a query.

```
SQL> ATTACH 'FILENAME personnel';
SQL> --
SQL> -- No optimization level has been selected.  The optimizer
SQL> -- selects the fast-first (FFirst) retrieval strategy to
SQL> -- retrieve the rows from the EMPLOYEES table in the
SQL> -- following query:
SQL> --
SQL> SELECT EMPLOYEE_ID, LAST_NAME
cont>  FROM EMPLOYEES
cont>  WHERE EMPLOYEE_ID IN ('00167', '00168');
Leaf#01 FFirst RDB$RELATIONS Card=19
  BgrNdx1 RDB$REL_REL_NAME_NDX [1:1] Fan=8
Sort
Cross block of 2 entries
  Cross block entry 1
    Leaf#01 BgrOnly RDB$RELATION_FIELDS Card=71
      BgrNdx1 RDB$RFR_REL_NAME_FLD_ID_NDX [1:1] Fan=8
  Cross block entry 2
    Get     Retrieval by index of relation RDB$FIELDS
      Index name  RDB$FIELDS_NAME_NDX [1:1]  Direct lookup
Leaf#01 FFirst EMPLOYEES Card=100
  BgrNdx1 EMP_EMPLOYEE_ID [1:1...]2 Fan=17
 EMPLOYEE_ID   LAST_NAME
 00167         Kilpatrick
 00168         Nash
2 rows selected
SQL> --
SQL> -- Use the SET OPTIMIZATION LEVEL statement to specify that
SQL> -- you want the total-time (BgrOnly) retrieval strategy to
SQL> -- be used.  Note that when the previous query is executed
SQL> -- again, the total-time (BgrOnly) retrieval strategy is
SQL> -- selected, instead of fast-first.
SQL> --
SQL> SET OPTIMIZATION LEVEL 'TOTAL TIME';
SQL> SELECT EMPLOYEE_ID, LAST_NAME
cont>  FROM EMPLOYEES
cont>  WHERE EMPLOYEE_ID IN ('00167', '00168');
Leaf#01 BgrOnly EMPLOYEES Card=100
  BgrNdx1 EMP_EMPLOYEE_ID [1:1...]2 Fan=17
 EMPLOYEE_ID   LAST_NAME
 00167         Kilpatrick
 00168         Nash
2 rows selected
SQL> --
SQL> -- When the SET OPTIMIZATION LEVEL 'DEFAULT' statement
SQL> -- is specified, either the fast-first or total-time
SQL> -- strategy will be selected.  The fast-first strategy
SQL> -- will be tried first, then total-time will be selected
SQL> -- if it will retrieve the rows faster than the fast-first
SQL> -- strategy.
SQL> --
SQL> SET OPTIMIZATION LEVEL 'DEFAULT';
SQL> --
SQL> -- Because the fast-first strategy is faster than the
SQL> -- total-time strategy for this query, the fast-first
SQL> -- stragegy is used to retrieve the rows.
SQL> --
SQL> SELECT EMPLOYEE_ID, LAST_NAME
cont>  FROM EMPLOYEES
cont>  WHERE EMPLOYEE_ID IN ('00167', '00168');
Leaf#01 FFirst EMPLOYEES Card=100
  BgrNdx1 EMP_EMPLOYEE_ID [1:1...]2 Fan=17
 EMPLOYEE_ID   LAST_NAME
 00167         Kilpatrick
 00168         Nash
2 rows selected
```

## SET QUOTING RULES Statement

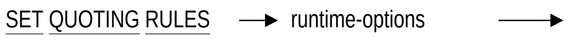
Specifies whether strings within double quotation marks are interpreted as string literals or delimited identifiers in the current connection.

### Environment

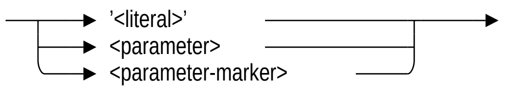
You can use the SET QUOTING RULES statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**′literal′**
**parameter**
**parameter-marker**
Specifies the value of the runtime-options, which must be one of the following:

- SQL92
- SQL89
- MIA
- SQLV40

**SQL92**
**SQL89**
**MIA**
Specifies that SQL interprets strings within double quotation marks as delimited identifiers. Delimited identifiers are case sensitive.

To comply with the ANSI/ISO SQL standard naming conventions, you should use the SQL92, SQL89, or MIA option. In addition, you must use one of these options to use multischema database naming.

**SQLV40**
Specifies that SQL interprets strings within double quotation marks as string literals.

The default is SQLV40.

## Usage Notes

- If the SET DIALECT statement is processed after the SET QUOTING RULES statement, it can override the setting of the SET QUOTING RULES statement.
- If the SET KEYWORD RULES statement is processed after the SET QUOTING RULES statement, it can override the setting of the SET QUOTING RULES statement.
- Specifying the SET QUOTING RULES statement changes the quoting rules for the current connection only. Use the SHOW CONNECTIONS statement to display the characteristics of a connection.

## Examples

Example 1: Setting the quoting rules to SQL92

```
SQL> SET QUOTING RULES ’SQL92’;
SQL> --
SQL> -- SQL interprets double quotation marks as delimited identifiers.
SQL> --
SQL> CREATE TABLE "Employees_Table"
cont>  ("Employee_ID" CHAR(6),
cont>    "Employee_Name" CHAR (30));
SQL> --
SQL> -- SQL retains the upper- and lowercase letters within the identifier.
SQL> --
SQL> SHOW TABLE EMPLOYEES_TABLE
No tables found
SQL> SHOW TABLE "Employees_Table"
Information for table Employees_Table

Columns for table Employees_Table:
Column Name                     Data Type        Domain
-----------                     ---------        ------
Employee_ID                     CHAR(6)
Employee_Name                   CHAR(30)

   .
   .
   .
```

Example 2: Setting the quoting rules to SQLV40

```
SQL> SET QUOTING RULES 'SQLV40';
SQL> --
SQL> -- When you set the quoting rules to SQLV40, SQL interprets double
SQL> -- quotation marks as string literals.
SQL> --
SQL> CREATE TABLE "Employees_Table"
%SQL-I-DEPR_FEATURE, Deprecated Feature: " used instead of ' for string
literal
CREATE TABLE "Employees_Table"
              ^
%SQL-W-LOOK_FOR_STT, Syntax error, looking for:
%SQL-W-LOOK_FOR_CON,           name, FROM,
%SQL-F-LOOK_FOR_FIN,    found Employees_Table instead
SQL> --
SQL> -- Although you can use double quotation marks for string literals, SQL
SQL> -- returns a deprecated feature message.
SQL> --
SQL> INSERT INTO EMPLOYEES
cont>       (EMPLOYEE_ID, LAST_NAME, STATUS_CODE)
cont> VALUES
cont>       ("00500", 'Toliver', '1');
%SQL-I-DEPR_FEATURE, Deprecated Feature: " used instead of ' for string
literal
1 row inserted
SQL> --
```

## SET SCHEMA Statement

Specifies the default schema name for an SQL user session in dynamically prepared and executed or interactive SQL statements until another SET SCHEMA statement is issued.

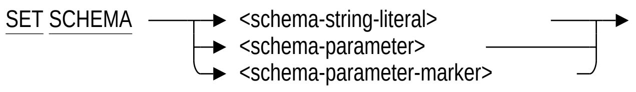
Within one multischema database, tables in different schemas can be used in a single SQL statement; tables in schemas in different databases cannot. If you omit the schema name when you specify an object in a multischema database, SQL uses the default schema name.

### Environment

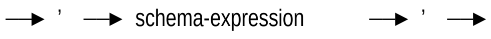
You can use the SET SCHEMA statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

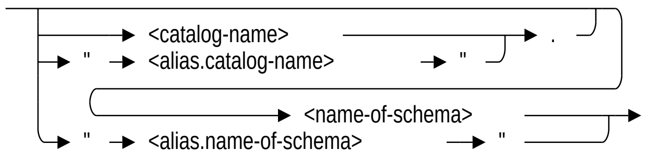
### Format



schema-string-literal =



schema-expression =



## Arguments

**schema-string-literal**
Specifies a character string literal that specifies the default schema. The schema string literal must contain a schema expression enclosed within single quotation marks.

**schema-parameter**
Specifies a host language variable in precompiled SQL or a formal parameter in an SQL module language procedure that specifies the default schema. The schema parameter must contain a schema expression.

**schema-parameter-marker**
Specifies a parameter marker (?) in a dynamic SQL statement. The schema parameter marker refers to a parameter that specifies the default schema. The schema parameter marker must specify a parameter that contains a schema expression.

**schema-expression**
Specifies the name of the default schema for a multischema database. If you omit the schema name when you specify an object in a multischema database, SQL uses the default schema name. If you do not specify a default schema name, the default is RDB$SCHEMA.

See Section 2.2.8 for more information on schemas.

## Usage Notes

- SQL does not issue an error message when you use SET SCHEMA to set default to a schema that does not exist. However, when you refer to that schema by specifying an unqualified name, SQL issues the error message shown in the following example:

```
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> SHOW CATALOGS
Catalogs in database CORP
    "CORP.ADMINISTRATION"
    "CORP.RDB$CATALOG"
SQL> SHOW SCHEMAS
Schemas in database with filename corporate_data
    ACCOUNTING
    PERSONNEL
    RECRUITING
    RDB$CATALOG.RDB$SCHEMA
SQL> SET SCHEMA '"CORP.ADMINISTRATION".BOGUS';
SQL> CREATE TABLE NEWTABLE (COL1 REAL);
%SQL-F-SCHNOTDEF, Schema BOGUS is not defined
```

Remember that the double-quoted leftmost pair (the delimited identifier) in a multischema object name requires uppercase characters. For other multischema naming rules, see Section 2.2.3. You will receive the following error message if you specify a delimited identifier in lowercase characters:

```
SQL> set schema '"corp.administration".accounting';
SQL> CREATE TABLE NEWTABLE (COL1 REAL);
%SQL-F-NODEFDB, There is no default database
SQL> set schema '"CORP.ADMINISTRATION".accounting';
SQL> CREATE TABLE NEWTABLE (COL1 REAL);
SQL>
```

- You cannot use the SET SCHEMA statement for nondynamic statements.

## Example

Example 1: Setting schema and catalog defaults to create a table in a multischema database

In this example, user ELLINGSWORTH attaches to two databases: the default database, personnel, and the multischema corporate_data database with alias CORP. User ELLINGSWORTH attempts to create a table in the corporate_ data database, and receives an error message because the default schema is ELLINGSWORTH, which has not been created in the default catalog. User ELLINGSWORTH uses SET SCHEMA and SET CATALOG statements to change the defaults to catalog ADMINISTRATION and schema ACCOUNTING of the corporate_data database.

Use the SHOW DATABASE statement to see the database settings.

```
SQL> ATTACH 'FILENAME personnel';
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL>  SHOW SCHEMAS;
Schemas in database with filename personnel
No schemas found
Schemas in database CORP
    "CORP.ADMINISTRATION".ACCOUNTING
    "CORP.ADMINISTRATION".PERSONNEL
    "CORP.ADMINISTRATION".RECRUITING
    "CORP.RDB$CATALOG".RDB$SCHEMA
SQL> CREATE TABLE CORP.BUDGET (COL1 REAL);
%SQL-F-SCHNOTDEF, Schema "CORP.RDB$CATALOG".CORP is not defined
SQL> --
SQL> -- SQL interprets CORP as schema name, and there is no
SQL> -- CORP schema in the default database.
SQL> --
SQL> -- Add quotation marks to designate qualifier CORP as an alias,
SQL> -- not the schema name.
SQL> --
SQL> SET QUOTING RULES 'SQL92';
SQL> CREATE TABLE "CORP.BUDGET" (COL1 REAL);
%SQL-F-SCHNOTDEF, Schema "CORP.RDB$CATALOG".ELLINGSWORTH is not defined
SQL> --
SQL> -- The default schema in the database with alias CORP
SQL> -- is the user name ELLINGSWORTH, but there is no
SQL> -- schema named ELLINGSWORTH.
SQL> --
SQL> -- Set the default schema to ACCOUNTING, and qualify it
SQL> -- with a delimited identifier containing the alias CORP and
SQL> -- the catalog ADMINISTRATION.  Now you can create the
SQL> -- table BUDGET within schema ACCOUNTING without qualifying
SQL> -- the table name.
SQL> --
SQL> SET SCHEMA '"CORP.ADMINISTRATION".ACCOUNTING';
SQL> CREATE TABLE BUDGET (COL1 REAL);
SQL> SHOW TABLES;
User tables in database with filename personnel
     CANDIDATES
     COLLEGES
     .
     .
     .
     User tables in database with alias CORP
     "CORP.ADMINISTRATION".ACCOUNTING.BUDGET
     .
     .
     .
```

## SET TRANSACTION Statement

Starts a transaction and specifies its characteristics. A **transaction** is a group of statements whose changes can be made permanent or undone only as a unit. The characteristics specified in a SET TRANSACTION statement affect all transactions until the transaction ends.

A transaction ends with a COMMIT or ROLLBACK statement. If you end the transaction with the COMMIT statement, all the changes made to the database by the statements are made permanent. If you end the transaction with the ROLLBACK statement, the statements do not take effect.

You must end the transaction with a COMMIT or ROLLBACK statement before starting or declaring another transaction. If you try to start or declare a transaction while another one is active, SQL generates an error message.

Besides the SET TRANSACTION statement, you can specify the characteristics of a transaction in one of two other ways:

- If you specify the DECLARE TRANSACTION statement, the declarations in the statement take effect when SQL starts a new transaction that is not started by the SET TRANSACTION statement. SQL starts a new transaction with the first executable data manipulation or data definition statement following the DECLARE TRANSACTION, COMMIT, or ROLLBACK statement.
- If you omit both the DECLARE and SET TRANSACTION statements, SQL automatically starts a transaction (using the read/write option) with the first executable data manipulation or data definition statement following a COMMIT or ROLLBACK statement. Thus, you can retrieve and update data without declaring or setting a transaction explicitly.

See the Usage Notes for examples of when you would want to use the DECLARE TRANSACTION statement instead of the SET TRANSACTION statement.

You can specify many options with the SET TRANSACTION statement, including:

- Transaction mode (READ ONLY/READ WRITE)
- Lock specification clause (RESERVING options)
- Wait mode (WAIT/NOWAIT)
- Isolation level
- Constraint evaluation specification clause

- Multiple sets of all the preceding options for each database involved in the transaction (ON . . . AND ON)

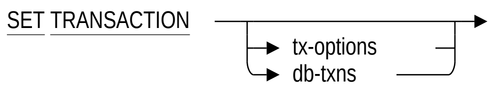
The Arguments section explains these options in more detail.

## Environment

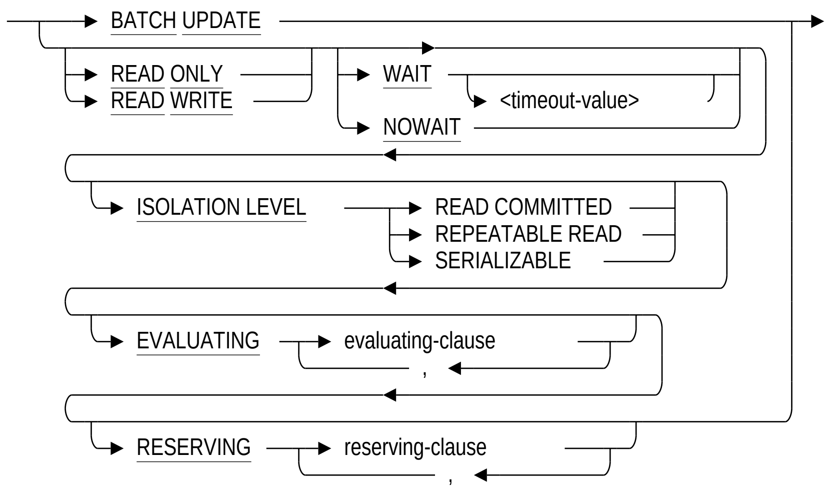
You can use the SET TRANSACTION statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

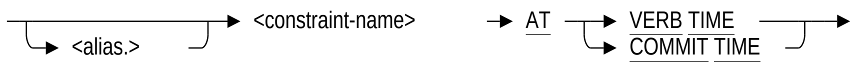
## Format



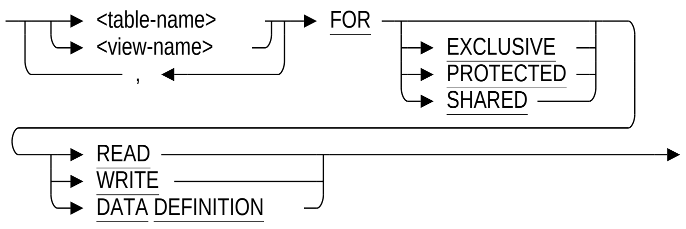
tx-options =



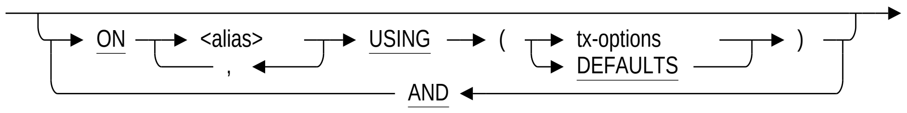
evaluating-clause =



reserving-clause =



db-txns =



## Arguments

**BATCH UPDATE**

Specifies the batch-update mode to reduce overhead in large-load operations. To speed update operations, Oracle Rdb does not write to snapshot or recovery-unit journal files in a batch-update transaction. For more information about batch-update transactions, see the *Oracle Rdb7 Guide to SQL Programming*.

Without the support of a recovery-unit journal file, you cannot roll back a batch-update transaction. If the load fails, you must create the database again. For example, if you need a large test database for development purposes, a batch-update transaction loads the database but bypasses the journaling facilities.

When you specify the batch-update transaction in your SET TRANSACTION statement, the load or update task results in access to the entire database. The batch-update transaction, for example, requires your transaction to be the only transaction accessing the database. It is the most efficient transaction you can choose when you load the entire database for the first time, or when you accumulate database updates in a data file that you intend to apply to the database at one time.

In addition to requiring the fewest locks on the database, the batch-update transaction permits updates to the database without creating a recovery-unit journal file. Therefore, any records changed or added during the batch-update transaction cannot be rolled back because Oracle Rdb does not maintain before-images of the changed records.

Because you cannot use batch-update transactions with distributed transactions, you should define the SQL$DISABLE_CONTEXT logical name or SQL_DISABLE_CONTEXT configuration parameter as "True" before you start a batch-update transaction. (Distributed transactions require that you are able to roll back transactions.)

If you attempt to start a distributed transaction using a batch-update transaction, what happens depends upon what transaction manager you are using, whether you call the DECdtm system services implicitly or explicitly, and which SQL statement you use to start the transaction.

Digital UNIX

- If you use the X/Open XA transaction manager and start a batch update transaction, Oracle Rdb returns an error at compile time. ♦

OpenVMS VAX OpenVMS Alpha

- If you start a batch-update transaction and explicitly call the DECdtm system services, SQL returns an error at compile time.
- If you start a batch-update transaction and implicitly call the DECdtm system services, SQL takes the following actions:
  - If you use a SET TRANSACTION statement with the BATCH UPDATE clause, SQL starts a non-distributed transaction.
  - If you use a DECLARE TRANSACTION statement with the BATCH UPDATE clause, SQL returns an error at compile time. ♦

The two-phase commit protocol applies only to distributed transactions. For more information about distributed transactions, see the *Oracle Rdb7 Guide to Distributed Transactions*.

A batch-update transaction started on a database cannot include additional arguments. However, other databases referred to in the same transaction declaration can include other arguments.

For example, the following statement is valid:

```
SQL> SET TRANSACTION ON OLD_DB USING (READ ONLY)
cont>    AND ON NEW_DB USING (BATCH UPDATE);
```

**Caution**

Before you begin a batch-update transaction in your programs, you should create a backup copy of the database using the RMU Backup command. If an error occurs in your program that would normally result in a rollback of the transaction, Oracle Rdb marks the database as corrupt. To recover from a corrupt database, you must create the database again from the backup copy of the database. After you corrected the error condition, you can restart the program from the beginning. You should back up the database after completing a batch-update transaction as well.

---

**READ ONLY**
Retrieves a snapshot of the database at the moment the read-only transaction starts. Other users can update rows in the table you are using, but your transaction retrieves the rows as they existed at the time the transaction started. You cannot update, insert, or delete rows, or execute data definition statements in a read-only transaction with the exception of declaring a local temporary table or modifying data in a created or declared temporary table. Read-only transactions do not have an isolation level.

Because a read-only transaction uses the snapshot (.snp) version of the database, any changes that other users make and commit during the transaction are invisible to you. Using a read-only transaction lets you read data without incurring the overhead of row locking. (You do incur overhead for keeping a snapshot of the tables you specify in the RESERVING clause, but this overhead is less than that of a comparable read/write transaction.)

Because of the limited nature of read-only transactions, they are subject to several restrictions. The Usage Notes describe those restrictions.

**READ WRITE**
Signals that you want to use the lock mechanisms of SQL for consistency in data retrieval and update. Read/write is the default transaction. Use the read/write transaction mode when you need to:

- Insert, update, or delete data
- Retrieve data that is guaranteed to be correct at the moment of retrieval
- Use SQL data definition statements

When you are reading a row in a read/write transaction, no other user can update that row. Under some circumstances, SQL may lock rows that you are not explicitly reading.

- If your query is scanning a table without using an index, SQL locks all the rows in the record stream.
- If your query uses indexes, SQL may lock part of an index, which has the effect of locking many rows.

**WAIT**
**NOWAIT**
Determines what your transaction does when it encounters a locked row. The default is WAIT.

- If you specify WAIT, the transaction waits for other transactions to complete and then proceeds (unless there is a deadlock, in which case you need to roll back the transaction). If you prefer, you can specify that the transaction proceeds after a certain time interval instead of waiting for other transactions to complete. You can specify the timeout interval value after the WAIT keyword. The timeout interval value is expressed in seconds.
- If you specify NOWAIT, your transaction returns an error message when it encounters a locked row.

**timeout-value**
Specifies the number of seconds for a given transaction to wait for other transactions to complete. This interval is only valid for the transaction specified in the SET TRANSACTION statement. Subsequent transactions return to the database default timeout interval. A timeout value of 0 specifies NOWAIT.

When you specify a wait interval in more than one way—any combination of logical name (RDM$BIND_LOCK_TIMEOUT_INTERVAL) or configuration parameter (RDB_BIND_LOCK_TIMEOUT_INTERVAL), application wait interval, and database parameter (using the CREATE DATABASE statement)— Oracle Rdb uses the lowest wait interval.

The following describes, in more detail, which wait interval takes precedence:

- If you specify an application wait interval, that interval overrides any wait interval specified by the logical name or configuration parameter that has a *higher* value.
- If you specify a database-wide wait interval (using the CREATE DATABASE statement), that interval overrides any other wait interval that has a *higher* value.

**ISOLATION LEVEL READ COMMITTED**
**ISOLATION LEVEL REPEATABLE READ**
**ISOLATION LEVEL SERIALIZABLE**
Defines the degree to which database operations in an SQL transaction are affected by database operations in concurrently executing transactions. It determines the extent to which the database protects the consistency of your data.

Oracle Rdb supports isolation levels READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. When you use SQL with Oracle Rdb databases, by default, SQL executes a transaction at isolation level SERIALIZABLE. The higher the isolation level, the more isolated a transaction is from other currently executing transactions. Isolation levels determine the type of phenomena that are allowed to occur during the execution of concurrent transactions. Two phenomena define SQL isolation levels for a transaction:

- Nonrepeatable read

  Allows the return of different results within a single transaction when an SQL operation reads the same row in a table twice. Nonrepeatable reads can occur when another transaction modifies and commits a change to the row between transaction reads.

- Phantom

  Allows the return of different results within a single transaction when an SQL operation retrieves a range of data values (or similar data existence check) twice. Phantoms can occur if another transaction inserted a new record and committed the insertion between executions of the range retrieval.

Each isolation level differs in the phenomena it allows. Table 7–7 shows the phenomena permitted for the isolation levels that you can explicitly specify with the SET TRANSACTION statement.

**Table 7–7 Phenomena Permitted at Each Isolation Level**

| Isolation Level | Nonrepeatable Reads Allowed? | Phantoms Allowed? |
|---|---|---|
| READ COMMITTED | Yes | Yes |
| REPEATABLE READ | No | Yes |
| SERIALIZABLE | No | No |

For read-only transactions, which always execute at isolation level SERIALIZABLE if snapshots are enabled, the database system guarantees that you will not see changes made by another user before you issue a COMMIT statement.

See the *Oracle Rdb7 Guide to SQL Programming* for further information about specifying isolation levels in transactions.

**evaluating-clause**
Specifies the point at which the named constraint or constraints are evaluated. If you specify VERB TIME, they are evaluated when the data manipulation statement is issued. If you specify COMMIT TIME, the constraint evaluation is based on the setting of the SET ALL CONSTRAINTS statement. For read-only transactions, this clause is allowed but is meaningless.

**alias**
Specifies the alias for a constraint. See the Usage Notes for information on using aliases for a multischema database.

**constraint-name**
Specifies the name of a constraint.

**RESERVING table-name**
**RESERVING view-name**
Lists the tables to be locked during the transaction. Include all the persistent base tables your transaction will access. You cannot reserve created or declared temporary tables.

If you use the RESERVING clause to specify tables, you can access only the tables you have reserved. However, specifying a view in a RESERVING clause is the same as specifying the base tables on which the view is based.

**FOR EXCLUSIVE**
**FOR PROTECTED**
**FOR SHARED**
Specifies the SQL share modes. The keyword you choose determines which operations you allow others to perform on the tables you are reserving. While you can specify an EXCLUSIVE or PROTECTED share mode when declaring a read-only transaction, SQL ignores these entries and specifies SHARED mode. The default is SHARED. Table 7–8 describes the different share modes.

**Table 7–8 SQL Share Modes**

| Option | Access Constraints |
|---|---|
| SHARED (Default) | Other users also can work with the same tables. Depending on the option they choose, they can have read-only or read/write access to the tables. |
| PROTECTED | Other users can read the tables you are using. They cannot have write access. |
| EXCLUSIVE | Other users cannot read records from the tables included in your transaction. If another user refers to the same tables in a DECLARE TRANSACTION statement, SQL denies access to that user. |

Under some circumstances, the base database system may promote a shared reservation to protected or exclusive to minimize locking of table rows.

Table 7–9 compares the effect of different lock specifications.

**Table 7–9 Comparison of Row Locking for Updates**

| Lock Specification | SHARED WRITE | PROTECTED WRITE | EXCLUSIVE WRITE | BATCH UPDATE |
|---|---|---|---|---|
| Writes to .ruj? | Yes | Yes | Yes | No |
| Writes to .snp? | Yes | Yes | No | No |
| Recovery? | Yes | Yes | Yes | No |
| Multiuser access? | Yes | Yes | No | No |

Table 7–10 compares the effects of different lock specifications on multiuser access.

**Table 7–10 Effects of Lock Specifications on Multiuser Access**

| For Tables You Reserve | Other Users Can Access the Tables | Your Effect on Other Users | Other Users' Effect on You |
|---|---|---|---|
| **READ WRITE** | | | |
| EXCLUSIVE READ<br>EXCLUSIVE WRITE<br>EXCLUSIVE DATA DEFINITION | No access | No one else can use the table. | No effect. |
| PROTECTED READ | PROTECTED READ<br>SHARED READ | No one else can write to the table. | No effect. |
| PROTECTED WRITE | SHARED READ | No one else can write to the table. No one else can read rows you use in any way until you end your transaction. | You cannot update rows other users read from a read/write transaction. |
| SHARED READ | PROTECTED READ<br>PROTECTED WRITE<br>SHARED READ<br>SHARED WRITE | A SHARED WRITE user cannot update rows you use in any way. | You cannot read rows that read/write transactions insert or update until those transactions end. |

(continued on next page)

**Table 7–10 (Cont.) Effects of Lock Specifications on Multiuser Access**

| For Tables You Reserve | Other Users Can Access the Tables | Your Effect on Other Users | Other Users' Effect on You |
|---|---|---|---|
| **READ WRITE** | | | |
| SHARED WRITE | SHARED READ SHARED WRITE | No one else can read or update rows you update. No one else can update rows you use in any way. | You cannot read or update rows that other read/write transactions use in any way. |
| SHARED DATA DEFINITION | SHARED DATA DEFINITION | No one can write or read from the reserved tables. Other users can define indexes concurrently if they issue the SHARED DATA DEFINITION clause. | No effect. |
| **READ ONLY** | | | |
| SHARED READ | All but EXCLUSIVE | No effect. | You do not see changes to rows. |

**READ**
**WRITE**
**DATA DEFINITION**
Specifies the lock type. These keywords declare what you intend to do with the tables you are reserving.

Use READ when you only want to read data from the tables. This is the default for read-only transactions.

Use WRITE when you want to insert, update, or delete data in the tables. This is the default for read/write transactions. You cannot specify WRITE for read-only transactions.

Use DATA DEFINITION when you want to create indexes at the same time other users are creating indexes, even if the indexes are on the same table. To allow concurrent index definition on the same table, use the SHARED DATA DEFINITION clause. This clause can be used only in read/write transactions. See the Usage Notes for additional information.

**db-txns**
Specifies different transaction options. When you attach to more than one database and want to specify different transaction options for each database, use this clause.

**ON alias**
**AND ON alias**
Specifies the alias for a database for which you want to specify transaction options. An alias is a name for a particular attach to a database. See the Usage Notes for information about using an alias with a multischema database.

Use the ON clause when you attach to more than one database and want to specify different transaction options for each database. (If you omit the ON clause, the single set of transaction options in the SET TRANSACTION statement applies to all attached databases.)

You can include multiple sets of transaction options, one for each database, in multiple ON clauses separated with the AND keyword. Example 3 illustrates a multiple-database transaction.

**USING (tx-options)**
**USING DEFAULTS**
Specifies the transaction options you want for the database referred to by the alias in the preceding ON clause. You can explicitly specify the transaction, wait mode, and isolation level option, or you can use the DEFAULTS keyword. Using DEFAULTS is equivalent to specifying READ WRITE WAIT plus the isolation level option appropriate for the database system you are using.

## Defaults

The SET TRANSACTION statement has several levels of defaults. If you omit the statement altogether or issue the SET TRANSACTION statement by itself, SQL sets a transaction READ WRITE WAIT ISOLATION LEVEL SERIALIZABLE.

In general, you should use explicit SET TRANSACTION statements, specifying READ WRITE or READ ONLY, a list of tables in the RESERVING clause, and a share mode and lock type for each table. The more specific you are in a SET TRANSACTION statement, the more efficient your database operations will be.

When a SET TRANSACTION statement starts a transaction, any unspecified transaction characteristics are normal SQL defaults. Table 7–11 summarizes the defaults for each option and combination of options.

**Table 7–11 Defaults for the SET and DECLARE TRANSACTION Statements**

| Option | Default |
|---|---|
| Transaction Mode:<br>• READ WRITE<br>• READ ONLY | The default is READ WRITE. Which transaction, if any, you specify determines the default lock specification. |

(continued on next page)

**Table 7–11 (Cont.) Defaults for the SET and DECLARE TRANSACTION Statements**

| Option | Default |
|---|---|
| Lock Specification: | |
| • RESERVING | • If you do not specify a transaction or a RESERVING clause, the default is SHARED WRITE.<br>• If you specify a read/write transaction and do not include a RESERVING clause, SQL determines the lock specification for each table when it is first accessed by a data manipulation statement. If the first reference to a table is within a read operation, the table is locked for SHARED READ. When the first update statement is issued, the table is locked for SHARED WRITE.<br>• If you specify a read/write transaction and include a RESERVING clause, the default is SHARED.<br>• If you do not specify a transaction but do include a RESERVING clause, the default is SHARED.<br>• If you specify a read-only transaction, the default is SHARED READ, whether or not you specify a RESERVING clause. |
| Share Mode: | The default is SHARED. |
| • SHARED | |
| • PROTECTED | |
| • EXCLUSIVE | |

(continued on next page)

**Table 7–11 (Cont.) Defaults for the SET and DECLARE TRANSACTION Statements**

| Option | Default |
|---|---|
| Lock Type:<br>• READ<br>• WRITE<br>• DATA DEFINITION | • If you specify a read/write transaction, the default is WRITE.<br>• If you specify a read-only transaction, the default, and only allowed lock type, is READ. |
| Concurrency Option:<br>• ISOLATION LEVEL READ COMMITTED<br>• ISOLATION LEVEL REPEATABLE READ<br>• ISOLATION LEVEL SERIALIZABLE | The default is ISOLATION LEVEL SERIALIZABLE. |
| Wait Mode:<br>• WAIT<br>• NOWAIT | The default is WAIT. |

## Usage Notes

- If an object is reserved PROTECTED or EXCLUSIVE, that table will not be subject to nonrepeatable reads (or phantoms) no matter what the isolation level of the transaction; however, the overall transaction can still experience these phenomena.
- When you use the SHARED DATA DEFINITION clause, no one (including you) can query or update the reserved table in the same transaction. Other users cannot perform any data definition operations on the reserved table other than creating indexes.
- To minimize lock conflicts with other users when using the SHARED DATA DEFINITION clause, commit the transaction immediately.

- All users who are defining indexes on the same table must reserve the table using the SHARED DATA DEFINITION clause.
- A RESERVING clause that specifies EXCLUSIVE access for the table will disable concurrent index definition, as only one user will be able to access the table.
- PROTECTED access cannot be declared with the DATA DEFINITION clause.
- A transaction with a RESERVING clause specifying the SHARED DATA DEFINITION option cannot appear in multistatement procedures.
- When using isolation level repeatable READ, you will find cases when Oracle Rdb holds long-term read locks on rows that are not really required to prevent the nonrepeatable read phenomenon. Isolation level REPEATABLE READ reduces index contention not data contention.
- When a sequential scan is done under isolation level READ COMMITTED, the number of lock operations performed will increase. If adjustable lock granularity is disabled, the number of lock operations (and the total number of locks) might increase to unacceptable levels.
- Read-only transactions use a snapshot of the database. For this reason, they are immune to interference from other transactions and are always serializable by default. The following SQL statements specify conflicting transaction options and, if specified, return an error message:

  ```
  SQL> SET TRANSACTION READ ONLY ISOLATION LEVEL READ COMMITTED;
  %SQL-F-SETTRASLI, SET TRANSACTION statement specifies conflicting options
  SQL> -- or
  SQL> SET TRANSACTION READ ONLY ISOLATION LEVEL REPEATABLE READ;
  %SQL-F-SETTRASLI, SET TRANSACTION statement specifies conflicting options
  ```

  The following statement is acceptable. However, it is deprecated and Oracle Rdb does not recommend using it:

  ```
  SQL> SET TRANSACTION READ ONLY CONSISTENCY LEVEL 2;
  ```

- For isolation level READ COMMITTED, if a row is read with a FOR UPDATE ONLY cursor, then the row is locked exclusively and the results will not change until a COMMIT or ROLLBACK statement is issued.
- If you reserve a table with a particular share mode, that share mode may override the behavior your specified isolation level implies. For example, nonrepeatable reads are always prevented in a table explicitly reserved for protected retrieval. Isolation level REPEATABLE READ will not gain you any additional concurrency in this case. If some tables are reserved for protected retrieval and others for concurrent retrieval, nonrepeatable

read prevention will not be attempted in the tables reserved for concurrent retrieval.

Thus, you can use interactions between the share mode locks and the isolation level to achieve specific aims; however, Oracle Rdb does not recommend this level of complexity be used for applications.

- The SET TRANSACTION statement is an executable statement that both specifies and starts one transaction. You can include multiple SET TRANSACTION statements in a host language source file or in an SQL language module (see Chapter 3). The SET TRANSACTION statement has the following advantages:
  - It gives you explicit control over when transactions are started.
  - It provides flexibility for changing transaction characteristics in a program source file.
- In contrast to the SET TRANSACTION statement, the DECLARE TRANSACTION statement is not executable and therefore does not start a transaction. (The declarations in a DECLARE TRANSACTION statement take effect when SQL starts an implicit transaction, that is, with the first executable data manipulation or data definition statement following the DECLARE TRANSACTION, COMMIT, or ROLLBACK statement.)

  You can specify only one DECLARE TRANSACTION statement in a host language source file or an SQL module language source file. The only way you can change transaction characteristics in programs using the DECLARE TRANSACTION statement (without using the SET TRANSACTION statement) is to put SQL statements in separate source files and specify different DECLARE TRANSACTION statements in each file.

  The advantages offered by the DECLARE TRANSACTION statement are:
  - It can establish transaction defaults for an interactive SQL session, a module or single host language file in a program, or any statements executed dynamically from a module. You might, for example, specify DECLARE TRANSACTION READ ONLY in the SQLINI.SQL file you create to set up your interactive SQL environment.

    In interactive SQL, the characteristics specified by a DECLARE TRANSACTION statement are valid until you enter another DECLARE TRANSACTION statement. (A COMMIT or ROLLBACK statement followed by a SET TRANSACTION statement may start a transaction with different characteristics, but subsequent transactions started implicitly will have the characteristics specified in the last DECLARE TRANSACTION statement.)

If you specify characteristics using a SET TRANSACTION statement, however, the characteristics apply only to that transaction. You must enter the statement again after every COMMIT or ROLLBACK statement to establish those characteristics again.

The following sequence shows a DECLARE TRANSACTION statement followed by a SET TRANSACTION statement. The SET TRANSACTION statement is followed by a ROLLBACK statement.

```
SQL> -- Declares characteristics for the first transaction:
SQL> --
SQL> DECLARE TRANSACTION READ WRITE;
SQL> --
SQL> -- There is no COMMIT or ROLLBACK statement between the
SQL> -- DECLARE and the SET statements:
SQL> --
SQL> SET TRANSACTION READ ONLY;
SQL> --
SQL> -- The ROLLBACK statement rolls back the SET TRANSACTION
SQL> -- statement.
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> -- The transaction characteristics are once again those
SQL> -- specified in the first DECLARE TRANSACTION statement:
SQL> --
SQL> SELECT * FROM EMPLOYEES;
```

– You can include the DECLARE TRANSACTION statement in an SQL context file.

  In the *Oracle Rdb7 Guide to SQL Programming*, the section about program transportability explains when you may need an SQL context file to support a program that includes SQL statements.

- If you want to control when your application attachments to databases, you can use explicit calls to the distributed transaction manager. For example, if your application starts a transaction using one database and performs some tasks, and then, based on the outcome of an operation, adds one of several databases to the transaction, explicitly calling the distributed transaction manager lets you control when your application attaches to the databases. For more information, see the *Oracle Rdb7 Guide to Distributed Transactions*.
- To prevent one database user from corrupting another user's picture of the database, SQL:
  – Delays an operation if the operation needs a row that is locked by another process, or returns an error if the user specified NOWAIT

  - Rejects an operation if deadlocks occur (where two processes have locked rows that each process needs)

  No part of a transaction that modifies a database is complete until the entire transaction is committed successfully. In particular, a deadlock may occur at any time during the transaction until it is successfully committed. In programs, except for transactions started in read-only or exclusive modes, you should check for RDB$_DEADLOCK after each database operation. In addition, your program should check for RDB$_LOCK_CONFLICT if the program declares a transaction NOWAIT.

  Generally, the best way to recover from a deadlock or lock conflict is to use a ROLLBACK statement and start the transaction again.

  When you insert or update data in shared mode, SQL may lock index nodes for indexes on that table. This feature ensures that SQL will be able to update those index nodes for the new data. This process frequently causes deadlocks.

- Because of the limited nature of read-only transactions, SQL imposes the following restrictions:
  - You cannot update, insert, or delete data, or execute data definition statements in a read-only transaction on persistent base tables. You can update, insert, or delete data in a read-only transaction on created or declared temporary tables. You can also declare a local temporary table in a read-only transaction.
  - In read-only transactions, you can specify only READ lock specifications. If you specify a WRITE lock specification, SQL generates an error.
  - Because a read-only transaction uses the snapshot (.snp) version of the database, SQL will not start a read-only transaction in a database created with the SNAPSHOT IS DISABLED argument. If you specify a read-only transaction for such a database, SQL implicitly declares a read/write transaction that reserves all tables for a shared read.
  - SQL considers the exclusive write lock specification incompatible with the read-only transaction mode because exclusive write transactions do not write changes to the snapshot version of the database. Read-only transactions cannot get an up-to-date snapshot of the database until the exclusive write transaction finishes.

If an update transaction reserves a table for exclusive write, and a subsequent read-only transaction by another user attempts to access that table and use the wait option (the default), the read-only transaction waits until the earlier exclusive write transaction commits or rolls back and then receives an error message. For example, assume that a user already has reserved the EMPLOYEES table for exclusive write. A second user enters:

```
SQL> ROLLBACK;
SQL> SET TRANSACTION READ ONLY WAIT;
SQL> SELECT * FROM EMPLOYEES;
[waits for EXCLUSIVE WRITE transaction to end]
       .
       .
       .
[EXCLUSIVE WRITE transaction performs COMMIT or ROLLBACK]

%RDB-E-LOCK_CONFLICT, request failed due to locked resource; no-wait
parameter specified for transaction
-RDMS-F-CANTSNAP, can't ready storage area for snapshots
```

The read-only transaction must issue the SELECT statement again after the error message.

If your transaction requires exclusive write access to an area of the database, you should be aware of the results of the exclusive write transaction on read-only transactions that try to access a copy of the same tables in the snapshot file.

- To use an alias with a multischema database, you must enable ANSI/ISO quoting rules and create a delimited identifier, as shown in Example 4. For more information about delimited identifiers, see Section 2.2.3.
- A process that enabled update carry-over locking at the table level can cause concurrency problems if the process reserves tables in PROTECTED READ or PROTECTED WRITE modes. Carry-over locking at the table level is set by defining the RDMS$AUTO_READY logical name or RDB_AUTO_READY configuration parameter. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information about logical name or configuration parameter and carry-over locking.
- If your application uses a server process that is attached to the database for long periods of time and causes the snapshot file to grow excessively, consider disabling prestarted transactions. (Prestarted transactions are enabled by default.) You can disable prestarted transactions using the PRESTARTED TRANSACTIONS ARE OFF clause of the ATTACH, CONNECT, DECLARE ALIAS, CREATE DATABASE, and IMPORT statements. For more information, see the ATTACH Statement and the *Oracle Rdb7 Guide to Database Performance and Tuning*.

- If you use the SET TRANSACTION statement in a stored procedure with either the RESERVING table clause or the EVALUATING constraint clause, SQL establishes procedure dependencies on the tables or constraints that you specify. See the CREATE MODULE Statement for a list of statements that can or cannot cause stored procedure invalidation.

  See the *Oracle Rdb7 Guide to SQL Programming* for detailed information about stored procedure dependency types and how metadata changes can cause invalidation of stored procedures.
- The SET TRANSACTION EVALUATING AT VERB TIME statement is not allowed for NOT DEFERRABLE constraints.
- Each table referenced by a view is automatically reserved in the same mode in which the view is reserved, unless the table is explicitly reserved in the SET TRANSACTION statement. In a READ ONLY transaction all tables are accessed for read-only.
- Any table referenced by a constraint or trigger is reserved in SHARED READ mode unless reserved at a higher mode by an explicit SET TRANSACTION statement.
- Any table updated by a trigger is reserved in SHARED WRITE mode, unless reserved at a higher mode by an explicit SET TRANSACTION statement. If the SET TRANSACTION statement has already reserved the table for READ access, an error is returned when the trigger is loaded.
- If a READ ONLY transaction is in progress, then neither triggers or constraints are active. Because triggers and constraints are loaded only for update operations, nothing is automatically reserved in this situation.
- Any table referenced in a COMPUTED BY column is not reserved. You must do this explicitly.

## Examples

Example 1: Starting a read-only transaction

```
SQL> SET TRANSACTION READ ONLY;
```

This statement lets you read data from the database but not insert or update data. When you retrieve data, you see the database records as they existed at the time SQL started the transaction. You do not see any updates to the database made after that time.

Example 2: Reserving specific tables with the SET TRANSACTION statement

The following statement lets you specify the intended action for each table in the transaction:

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> SET TRANSACTION READ WRITE RESERVING
cont>      EMPLOYEES FOR PROTECTED WRITE,
cont>      JOBS, SALARY_HISTORY FOR SHARED READ;
```

Assume that this transaction updates the EMPLOYEES table based on values found in two other tables: JOBS and SALARY_HISTORY.

- The transaction must update the EMPLOYEES table, so EMPLOYEES is readied for protected write access.
- The program will only read values from the JOBS and SALARY_HISTORY tables, so there is no need for write access or protected write access. However, you do intend to update records in the transaction, so the read-only transaction is not appropriate.

Example 3: Specifying multiple databases in a SET TRANSACTION statement

You can access multiple databases from within the same transaction. This example explains how you can benefit from this feature.

Because read-only access uses a snapshot (.snp) version of the database, you might encounter inconsistencies in the data your application retrieves. Therefore, if your program accesses one database using read-only, another transaction using read/write might be updating a table in the database root file (.rdb) itself.

The data your program reads from the snapshot file represents a before-image of the database record that the other program is updating. If you require absolute data consistency for certain update applications, you should specify read/write access for both databases, and permit other users to read one of the databases by including the shared read share mode. In this way, you maintain data consistency during updates, while permitting concurrent data retrieval from the database that your program reads.

However, any read/write transaction you set offers reduced concurrent access when compared to read-only access. For that reason, use read/write transactions only when necessary.

Before you can use the multiple database feature of the SET TRANSACTION statement, you must issue a DECLARE ALIAS statement that specifies each database you intend to access. The DECLARE ALIAS statement must include an alias. For example, the following DECLARE ALIAS statements identify two databases required by an update application:

```
EXEC SQL
DECLARE DB1 ALIAS FOR FILENAME personnel;
END EXEC
```

```
EXEC SQL
DECLARE DB2 ALIAS FOR FILENAME benefits;
END EXEC
```

Because the program needs to only read the EMPLOYEES table of the personnel database (DB1) but needs to change values in two tables (TUITION and STATUS) in the BENEFITS database, the update program might contain the following SET TRANSACTION statement:

```
EXEC SQL  SET TRANSACTION
             ON DB1 USING ( READ ONLY
               RESERVING DB1.EMPLOYEES FOR SHARED READ )
            AND
             ON DB2 USING ( READ WRITE
               RESERVING DB2.TUITION FOR SHARED WRITE
                         DB2.STATUS  FOR SHARED WRITE )
END EXEC
```

Example 4: Specifying a multischema database in a SET TRANSACTION statement

If one of the databases you access is a multischema database, you must specify it using a delimited identifier. The following example shows how to access the single-schema personnel database and the multischema corporate_data database. The table EMPLOYEES is located within the schema PERSONNEL in the catalog ADMINISTRATION within the CORPORATE_DATA database.

```
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL> SET QUOTING RULES 'SQL92';
SQL> SET CATALOG '"CORP.ADMINISTRATION"';
SQL> SET SCHEMA '"CORP.ADMINISTRATION".PERSONNEL';
SQL> --
SQL> SET TRANSACTION ON CORP USING (READ ONLY
cont>    RESERVING "CORP.EMPLOYEES" FOR SHARED READ)
cont>    AND ON PERS USING (READ WRITE RESERVING
cont>    PERS.EMPLOYEES FOR SHARED WRITE);
```

Example 5: Specifying evaluation at verb time in a SET TRANSACTION statement

The following example shows an insert into the DEGREES table of a newly acquired degree for EMPLOYEE_ID 00164. The new degree, MME, is evaluated and, because it is not one of the acceptable degree codes, an error message is returned immediately.

```
SQL> ATTACH 'FILENAME personnel';
SQL> SET TRANSACTION READ WRITE
cont>   EVALUATING DEGREES_FOREIGN1 AT VERB TIME,
cont>   DEGREES_FOREIGN2 AT VERB TIME,
cont>   DEG_DEGREE_VALUES AT VERB TIME
cont>   RESERVING DEGREES FOR PROTECTED WRITE,
cont>   COLLEGES, EMPLOYEES FOR SHARED READ;
SQL> SHOW TRANSACTION
Transaction information:
    Statement constraint evaluation is off

On the default alias
Transaction characteristics:
        Read Write
        Evaluating constraint DEGREES_FOREIGN1 at verb time
        Evaluating constraint DEGREES_FOREIGN2 at verb time
        Evaluating constraint DEG_DEGREE_VALUES at verb time
        Reserving table DEGREES for protected write
        Reserving table COLLEGES for shared read
        Reserving table EMPLOYEES for shared read
Transaction information returned by base system:
a read-write transaction is in progress
  - updates have not been performed
  - transaction sequence number (TSN) is 153
  - snapshot space for TSNs less than 153 can be reclaimed
  - session ID number is 21
SQL> INSERT INTO DEGREES
cont>   (EMPLOYEE_ID, COLLEGE_CODE, YEAR_GIVEN,
cont>    DEGREE, DEGREE_FIELD)
cont> VALUES
cont>   ('00164', 'PRDU',  1992,
cont>   'MME',  'Mech Enging');
%RDB-E-INTEG_FAIL, violation of constraint DEG_DEGREE_VALUES caused
operation to fail
-RDB-F-ON_DB, on database DISK1:[JONES.PERSONNEL]PERSONNEL.RDB;1
SQL> ROLLBACK;
```

Example 6: Explicitly setting isolation levels in a transaction

This statement lets you read data from and write data to the database. It also sets the transaction to run at isolation level READ COMMITTED, not at the higher default isolation level SERIALIZABLE.

```
SQL> SET TRANSACTION READ WRITE ISOLATION LEVEL REPEATABLE READ;
```

Example 7: Creating index concurrently

The following example shows how to reserve the table for shared data definition and how to create an index:

```
SQL> SET TRANSACTION READ WRITE
cont>    RESERVING EMPLOYEES FOR SHARED DATA DEFINITION;
SQL> --
SQL> CREATE INDEX EMP_LAST_NAME1 ON EMPLOYEES (EMPLOYEE_ID);
SQL> --
SQL> -- Commit the transaction immediately.
SQL> --
SQL> COMMIT;
```

## SET VIEW UPDATE RULES Statement

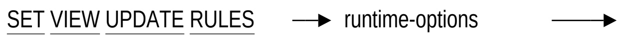
Specifies whether or not SQL applies the ANSI/ISO SQL standard for updatable views to views created during a session.

### Environment

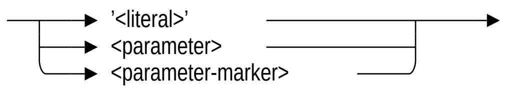
You can use the SET VIEW UPDATE RULES statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format



runtime-options



### Arguments

**'literal'**
**parameter**
**parameter-marker**
Specifies the value of runtime-options, which must be one of the following:

- SQL92
- SQL89
- MIA
- SQLV40

**SQL92**
**SQL89**
**MIA**
Specifies that the ANSI/ISO SQL standard for updatable views is applied to all views created during compilation. Views that do not comply with the ANSI/ISO SQL standard for updatable views cannot be updated.

The ANSI/ISO SQL standard for updatable views requires the following conditions to be met in the SELECT statement:

- The DISTINCT keyword is not specified.
- Only column names can appear in the select list. Each column name can appear only once. Functions and expressions such as max(column_name) or column_name +1 cannot appear in the select list.
- The FROM clause refers to only one table. This table must be either a base table or a derived table that can be updated.
- The WHERE clause does not contain a subquery.
- The GROUP BY clause is not specified.
- The HAVING clause is not specified.

**SQLV40**
Specifies that the ANSI/ISO SQL standard for updatable views is not applied.

SQL considers views that meet the following conditions to be updatable:

- The DISTINCT keyword is not specified.
- The FROM clause refers to only one table. This table must be either a base table or a derived table that can be updated.
- The WHERE clause does not contain a subquery.
- The GROUP BY clause is not specified.
- The HAVING clause is not specified.

The default is SQLV40.

## Usage Notes

- If the SET DIALECT statement is processed after the SET VIEW UPDATE RULES statement, it can override the setting of the SET VIEW UPDATE RULES statement.
- Specifying the SET VIEW UPDATE RULES statement changes the view rules for the current connection only. Use the SHOW CONNECTIONS statement to display the characteristics of a connection.

## Example

Example 1: Setting the view characteristics from SQLV40 to SQL92

```
SQL> ATTACH 'ALIAS MIA1 FILENAME MIA_CHAR_SET';
SQL> CONNECT TO 'ALIAS MIA1 FILENAME MIA_CHAR_SET' AS 'TEST';
SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: SQLV40
Default DATE type: DATE VMS
Quoting Rules: SQLV40
Optimization Level: DEFAULT
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

```
Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
SQL> --
SQL> -- Change the environment for view rules from SQLV40 to SQL92
SQL> --
SQL> SET VIEW UPDATE RULES 'SQL92';
SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: ANSI/ISO
Default DATE type: DATE VMS
Quoting Rules: SQLV40
Optimization Level: DEFAULT
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
```

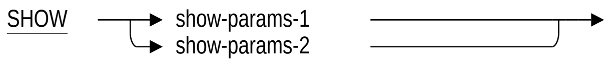
## SHOW Statement

Displays information about database entities and information about the interactive SQL session.

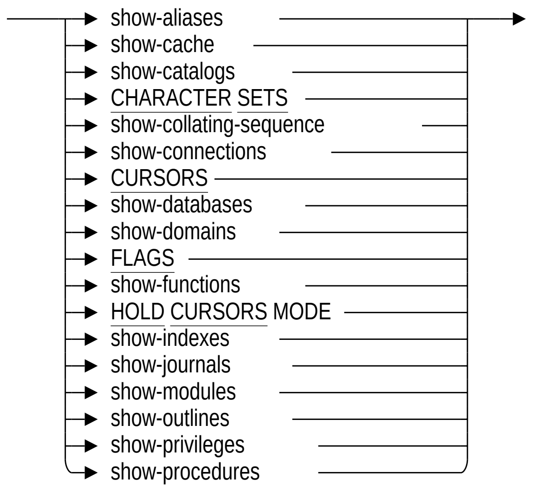
### Environment

You can use the SHOW statement only in interactive SQL.

### Format

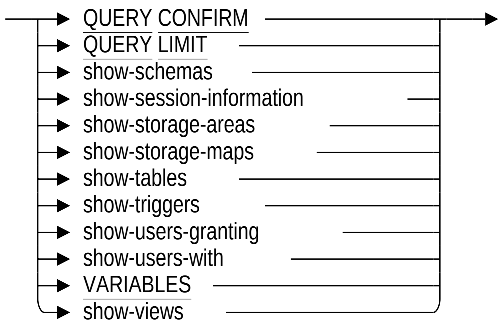


show-params-1 =

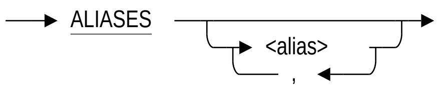


## SHOW Statement

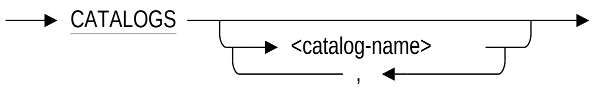
show-params-2 =



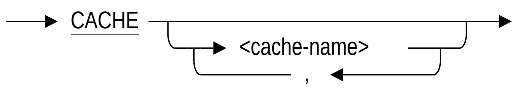
show-aliases =



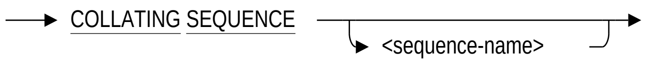
show-catalogs =



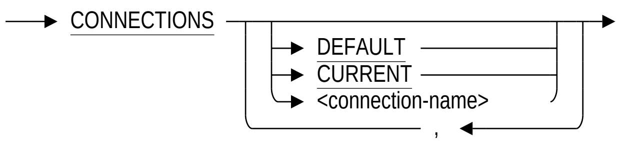
show-cache =



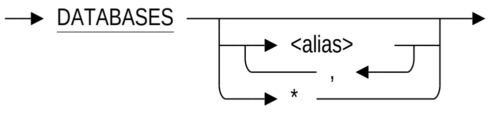
show-collating-sequence =



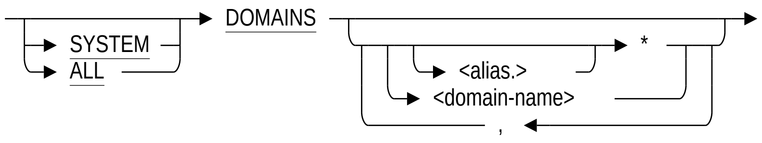
show-connections =



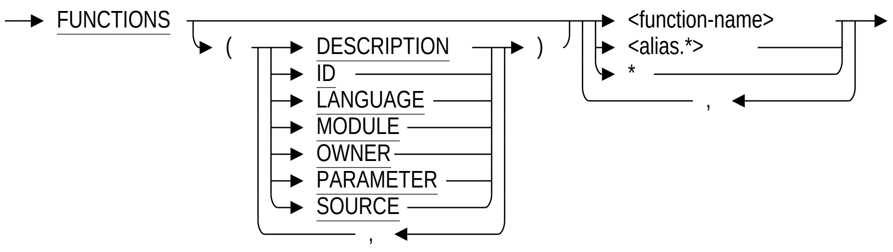
show-databases =



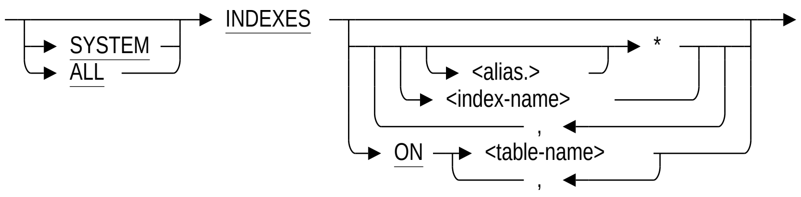
show-domains =



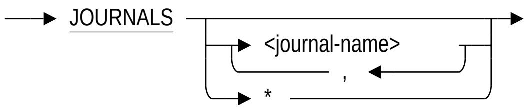
show-functions =



show-indexes =

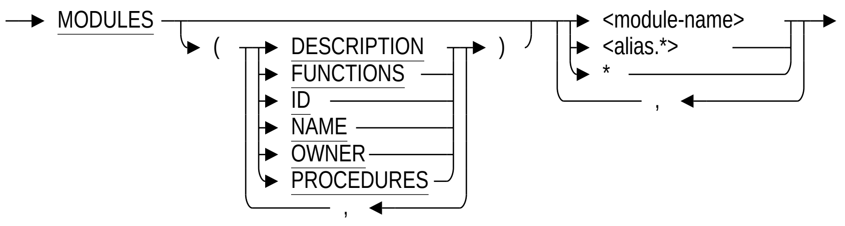


show-journals

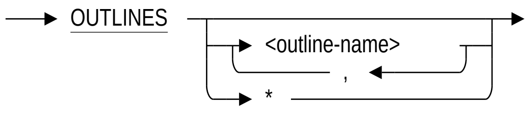


## SHOW Statement

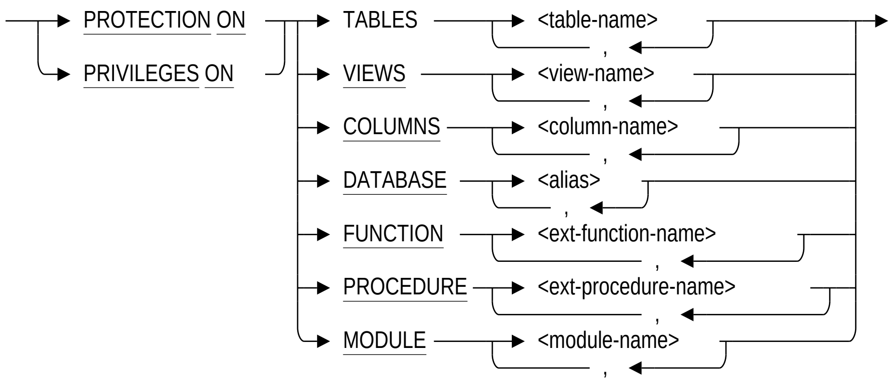
show-modules =



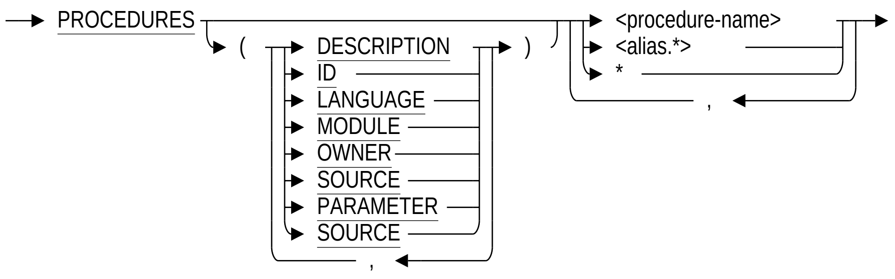
show-outlines



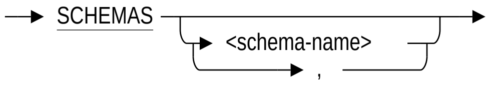
show-privileges =



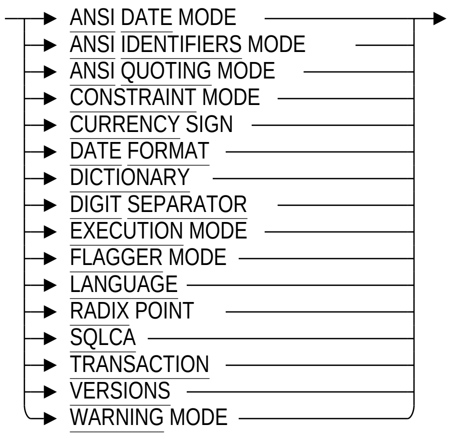
show-procedures =



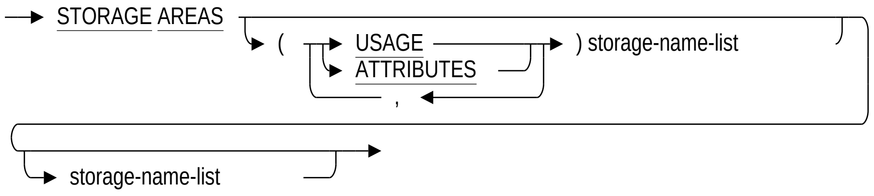
show-schemas =



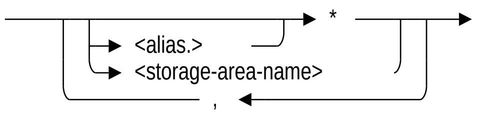
show-session-information =



show-storage-areas =

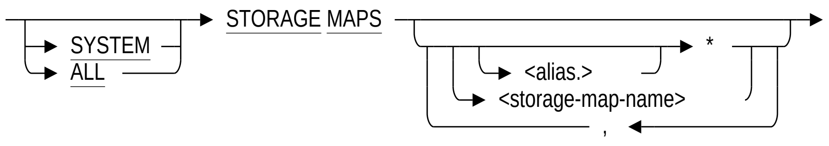


storage-name-list =

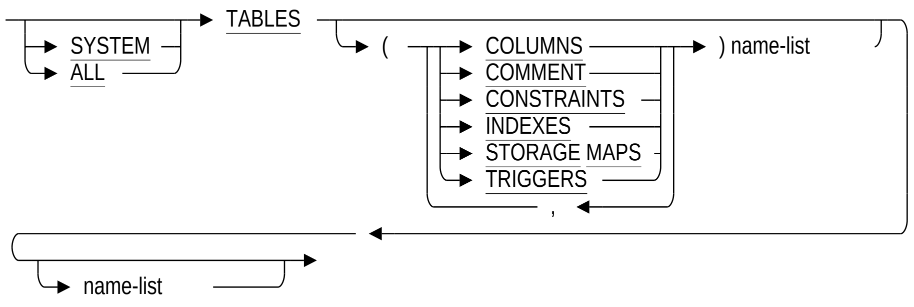


## SHOW Statement

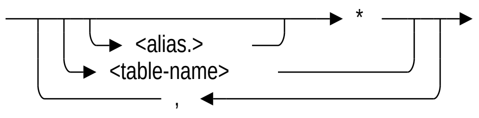
show-storage-maps =



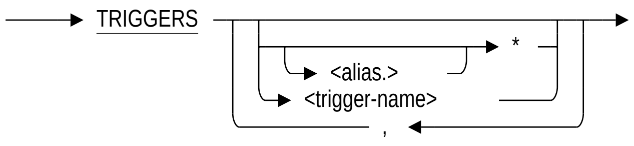
show-tables =



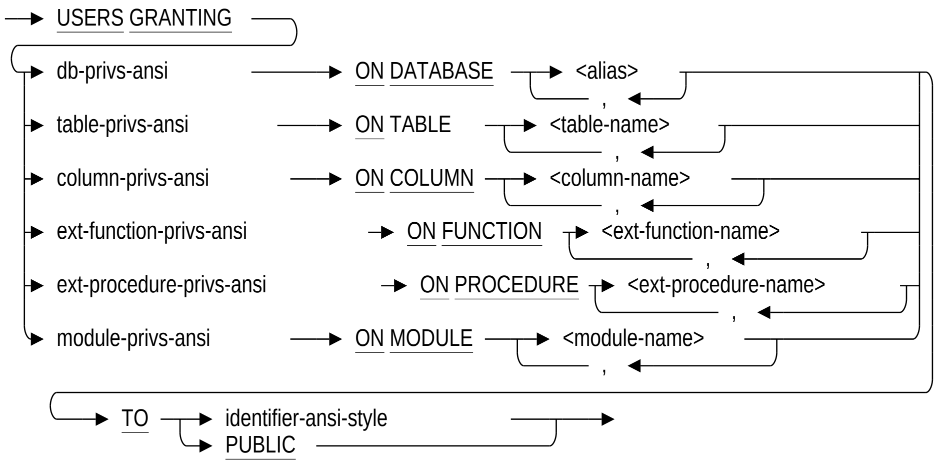
name-list =



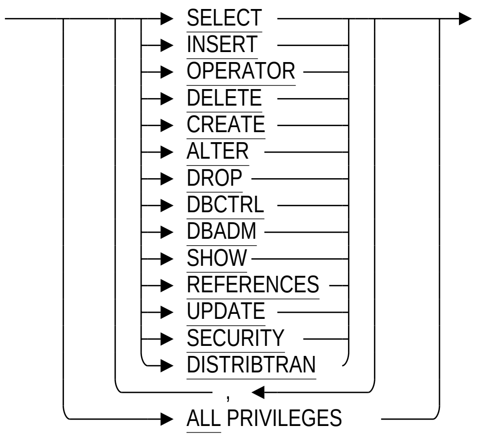
show-triggers =



show-users-granting =

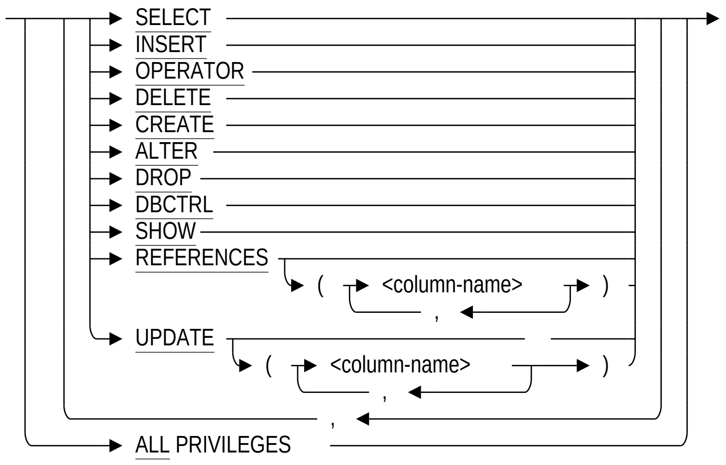


db-privs-ansi =

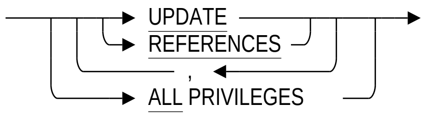


## SHOW Statement

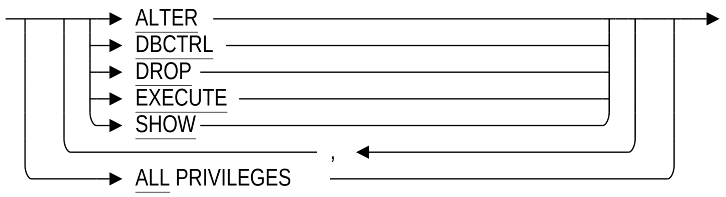
table-privs-ansi =



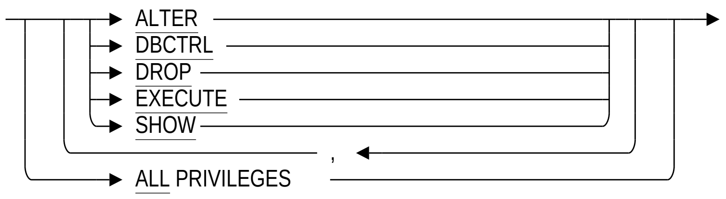
column-privs-ansi =



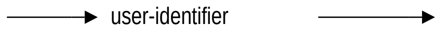
module-privs-ansi =



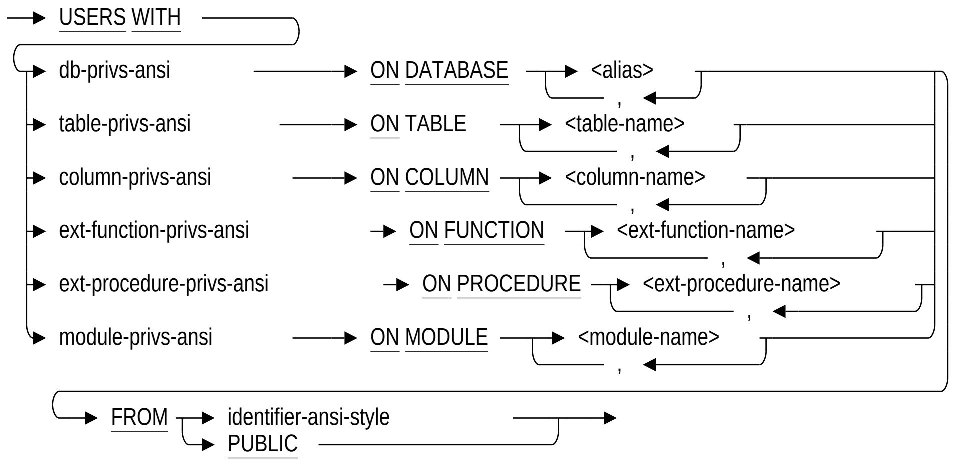
ext-routine-privs-ansi =



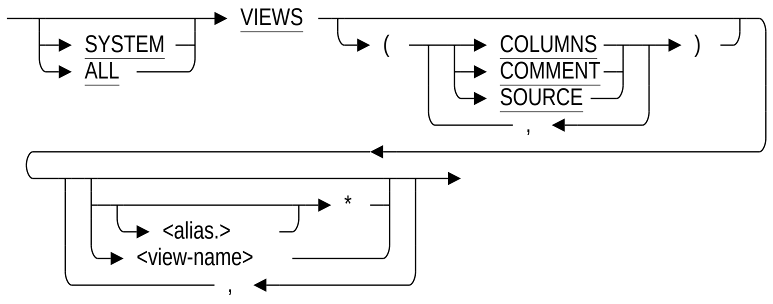
identifier-ansi-style =



show-users-with =



show-views =



## Arguments

**SYSTEM**
**ALL**
Controls whether SQL displays system-defined domains, indexes, storage maps, tables, or views in the SHOW DOMAINS, SHOW INDEXES, SHOW STORAGE MAPS, SHOW TABLES, and SHOW VIEWS statements.

- If you do not specify either SYSTEM or ALL, the display includes only user-defined elements.
- If you specify SYSTEM, the display includes only system-defined elements.
- If you specify ALL, the display includes both user-defined and system-defined elements.

#### ALIASES

Displays information about aliases for all attached databases. For each alias, SQL displays the path name or file name of the current default database, and the file specification for the database file.

If you specify aliases by name, SQL displays information about whether or not multischema mode, snapshots, carry-over locks, adjustable lock granularity, global buffers, commit to journal optimization, and journal fast commit are enabled. SQL displays the character sets of the alias if the database default, national, or identifier character set differs from the session's default, national, or identifier character set. SQL also displays the journal fast commit checkpoint and transaction intervals, the lock timeout interval, the number of users, number of nodes, buffer size, number of buffers, number of recovery buffers, ACL-based protections, storage areas, and whether or not the repository is required.

#### CACHE

Displays information about the specified cache. For example:

```
SQL> SHOW CACHE
Cache Objects in database with filename sample
        CACHE1
        CACHE2
SQL> SHOW CACHE cache1

     CACHE1
        Cache Size:            1000 rows
        Row Length:            256 bytes
        Row Replacement:       Enabled
        Shared Memory:         Process
        Large Memory:          Disabled
        Window Count:          100
        Reserved Rows:         20
        Sweep Rows:            3000
        No Sweep Thresholds
        Allocation:            100 blocks
        Extent:                100 blocks
```

**CATALOGS**
Displays information about the specified catalogs. If you do not specify any aliases in the catalog names that you specify, SQL displays this information about all attached databases.

**CHARACTER SETS**
Displays information about the specified character sets for the session and all attached databases.

**COLLATING SEQUENCE sequence-name**
Displays the collating sequences for schemas and domains.

**CONNECTIONS DEFAULT**
**CONNECTIONS CURRENT**
**CONNECTIONS connection-name**
Displays database information for the specified connection.

**CURSORS**
Displays current cursors.

**DATABASES**
Displays information about the specified databases. For each database, SQL displays the alias, the type of database, any defined collating sequence, and the file specification for the database file.

OpenVMS VAX OpenVMS Alpha

If the database was declared using a repository path name, SQL also displays that path name. If you do not specify any aliases with the SHOW DATABASES statement, SQL displays this information about all declared databases. ♦

SQL displays the character sets of the database if the default, national, or identifier character set differs from the session's default, national, or identifier character set.

If you do specify an alias, SQL also displays information about whether or not multischema mode, snapshots, carry-over locks, adjustable lock granularity, global buffers, commit to journal optimization, journaling, and journal fast commit are enabled. SQL also displays the journal fast commit checkpoint and transaction intervals, the lock timeout interval, the number of unused storage areas, the number of unused journal files, the number of users, number of nodes, buffer size, number of buffers, number of recovery buffers, ACL-based protections, storage areas, and whether or not the repository is required.

**DOMAINS**
Displays the names, data types, and character sets of specified domains. If you specify the SHOW DOMAINS statement without any arguments, SQL displays names, data types, and character sets of all domains in all attached databases.

*****
**alias.***
An asterisk wildcard, preceded by an optional alias. In a SHOW DOMAINS statement, specifying a wildcard displays the name, data type, formatting clauses, and comments for domains. If you do not precede the wildcard with an alias, SQL displays information about domains in the default database. If you precede the wildcard with an alias, SQL displays information about domains in that database.

**domain-name**
Specifies the name of a domain whose definition you want to display. SQL displays the name, data type, formatting clauses, collating sequence, and comments for each domain you specify.

**FLAGS**
Displays the database system debug flags that are enabled for the current session.

**FUNCTIONS**
Displays information about a specified function; either external or stored. When you enter the SHOW FUNCTIONS statement without any arguments, SQL displays the name of the function only. The following table lists the information that you can display using a set of keywords with the SHOW FUNCTIONS statement:

| You Specify This: | SQL Displays Information About: |
|---|---|
| DESCRIPTION | The description of the function. If none exists, nothing displays. |
| ID | The unique identification assigned to the function. |
| LANGUAGE | The host language in which the function is coded. |
| MODULE | The name of the module in which the function is defined. |
| OWNER | The owner of the function. |

| You Specify This: | SQL Displays Information About: |
|---|---|
| PARAMETER | Information about the parameters, including the number of arguments, the data type, return type, and how the parameter is passed. |
| SOURCE | Displays the source definitions for the specified functions. |

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. If you do not precede the wildcard with an alias, SQL displays information about the function in the default database. If you precede the wildcard with an alias, SQL displays information about functions in that database.

**HOLD CURSORS MODE**
Displays the default mode for hold cursors. For example:

```
SQL> SHOW HOLD CURSORS MODE
Hold Cursors default: WITH HOLD PRESERVE NONE
```

**INDEXES**
Displays information about specified indexes. SQL displays the name of the index, the associated column and table, the size of the index key, if the definition allows duplicate values for the column, the type of index (sorted or hashed), and whether index compression is enabled or disabled. If you specify the SHOW INDEXES statement without any arguments, SQL displays definitions of all indexes in all declared databases.

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. If you do not precede the wildcard with an alias, SQL displays information about indexes in the default database. If you precede the wildcard with an alias, SQL displays information about indexes in that database.

**index-name**
Specifies the name of an index whose definition you want to display.

**ON table-name**
Specifies the table or tables for which you want to see associated index definitions.

### JOURNALS

Displays information about specified journal files. SQL displays the name of the file specification and, if created, the backup file specification.

### MODULES

Displays information about specified modules.

If you do not specify any of the SHOW MODULES options listed in the following table, SQL displays information about all these options:

| You Specify This: | SQL Displays Information About: |
|---|---|
| DESCRIPTION | The description of the module. If none exists, nothing displays. |
| FUNCTIONS | The stored functions contained in the module. |
| ID | The unique identification assigned to the module. |
| NAME | The name of the module. |
| OWNER | The owner of the module. If the module is an invoker's rights module, this file is set to NULL and does not display anything. If the module is a definer's rights module, the definer's user name displays. |
| PROCEDURES | The stored procedures contained in the module. |

### *
### alias.*

Specifies an asterisk wildcard, preceded by an optional alias. If you do not precede the wildcard with an alias, SQL displays information about the module in the default database. If you precede the wildcard with an alias, SQL displays information about modules in that database.

### OUTLINES

Displays the definition of the specified outline. SQL displays the outline name, ID number, mode, query, compliance, and comment if one exists.

If you issue the SHOW OUTLINE statement without the name of a specific outline, the names of all the outlines stored in the database are displayed. However, the invalid outlines are not marked as invalid.

### PRIVILEGES
### PROTECTION

Displays current user identifier and available access rights for the specified object.

- The SHOW PRIVILEGES statement displays the current user identifier and available access rights to the specified databases, tables, views, columns, external functions, external procedures, or modules.

  This statement displays not only the privileges that were explicitly granted to the user, but also any privileges that the user inherits from database access or the system. For example, if, on Digital UNIX, the user attaches as the superuser dbsmgr, SQL shows that the user has all privileges, even though dbsmgr has not been explicitly granted any privileges on the database.

  In a client/server environment, the entry shows the identifier of the client. For example, if a user attaches to a remote database using the USER and USING clauses, SQL shows the privileges for the user specified in those clauses.

  In an environment that is not client/server, such as when you attach to a local database on OpenVMS, SQL shows not only the privileges of the database user, but of the logged-on process. For example, if user heleng, with the OpenVMS privilege BYPASS, uses the USER and USING clauses to attach to the database as user rhonda, SQL shows that user rhonda has the privileges inherited from the logged-on process heleng, as well as privileges for user rhonda.
- The SHOW PROTECTION statement displays all the entries in the access privilege set for the specified databases, tables, views, columns, external functions, external procedures, or modules.

**ON TABLES table-name**
**ON VIEWS view-name**
**ON COLUMNS column-name**
**ON FUNCTIONS ext-function-name**
**ON PROCEDURES ext-procedure-name**
**ON MODULES module-name**
Specifies the table, view, column, external function, external procedure, or module for which you want to display access privilege set information with the SHOW PRIVILEGES or SHOW PROTECTION statement. You can specify a list of tables, views, columns, external functions, external procedures, or modules, but you must specify at least one item to display a list. You must qualify a column name with at least the associated table name.

In an ANSI/ISO-style database, the SHOW PROTECTION statement displays which privileges have the option of being granted to other users and which privileges are without the grant option. See the SHOW USERS WITH and SHOW USERS GRANTING statements in this section for more information about displaying privileges granted directly or indirectly to other users.

**ON DATABASE alias**
Specifies the databases for which you want to display access privilege set information with the SHOW PRIVILEGES or SHOW PROTECTION statement. You can specify a list of aliases, but you must specify at least one. To display privileges for the default database, use the alias RDB$DBHANDLE.

**PROCEDURES**
Displays information about a specified procedure; either external or stored.

If you do not specify any of the SHOW PROCEDURES attributes (DESCRIPTION, ID, LANGUAGE, MODULE, OWNER, SOURCE, or PARAMETER), by default you will see the display for all these options.

| You Specify This: | SQL Displays Information About: |
|---|---|
| DESCRIPTION | The description of the stored procedure. If none exists, nothing displays. |
| ID | The unique identification assigned to the procedure. |
| LANGUAGE | The host language in which the procedure is called. |
| MODULE | The identification number of the module to which a procedure belongs. |
| OWNER | The owner of the procedure. |
| PARAMETER | Information about the parameters; including the number of arguments, the data type, and how the parameter is passed. |
| SOURCE | Displays the source definitions for the specified procedures. |

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. If you do not precede the wildcard with an alias, SQL displays information about the procedure in the default database. If you precede the wildcard with an alias, SQL displays information about procedures in that database.

**QUERY CONFIRM**
Shows whether or not SQL displays the cost estimates for a query before executing that query.

**QUERY LIMIT**
Displays information about the number of rows a query can return and the amount of time used to optimize a query for execution.

**SCHEMAS**
Displays the names of specified schemas. If you do not specify an alias as part of a schema name, SQL displays schema information for all the attached databases. For each database that is not multischema, SQL displays the message, “No schemas found”. For each multischema database, SQL displays the alias, followed by a list of schemas contained in that database. Each schema name in the list is preceded by the catalog and alias names.

**show session information**
Displays information about the current defaults set for the interactive SQL session.

**ANSI DATE MODE**
Displays the default interpretation for columns with the DATE or CURRENT_ TIMESTAMP data type.

The DATE and CURRENT_TIMESTAMP data types, can be either OpenVMS or SQL92. By default, both data types are interpreted as OpenVMS.

Use the SET DEFAULT DATE FORMAT statement to change the default date.

**ANSI IDENTIFIERS MODE**
Displays whether or not identifier checking is enabled. You must enclose reserved words from the ANSI/ISO SQL 1989 standard and the ANSI/ISO SQL standard within double quotation marks to supply them as identifiers in SQL statements. When you enable identifier checking, SQL issues an informational message after statements that misuse ANSI/ISO reserved words. For a list of the reserved words, see Section F.4.

By default, identifier checking is disabled. To enable it, use the SET KEYWORD RULES statement.

**ANSI QUOTING MODE**
Displays whether or not you must use double quotation marks to delimit the alias and catalog name pair in subsequent statements. By default, SQL syntax allows only single quotation marks. To comply with ANSI/ISO SQL standard naming conventions, you should set quoting rules to SQL92, SQL89, or MIA. You must set quoting rules to SQL92, SQL89, or MIA to use multischema database naming.

Use the SET QUOTING RULES statement to change the quoting rules.

**CONSTRAINT MODE**
Displays the default setting for constraint evaluation for any transactions starting after the current transaction. If there is a current transaction, displays the constraint evaluation mode for the current transaction.

When the constraint mode is IMMEDIATE, SQL evaluates all commit-time constraints at the end of each statement and at commit time, until the transaction completes or until you set the constraint mode to OFF. When the constraint mode is DEFERRED (the default setting), constraint evaluation is deferred until commit time.

**CURRENCY SIGN**
Displays the currency indicator, such as the dollar sign ( $ ), that will be used in output displays.

**DATE FORMAT**
OpenVMS VAX OpenVMS Alpha Displays the values for the date-number and time-number arguments of the SET DATE FORMAT DATE date-number and SET DATE FORMAT TIME time-number statements. This display is disabled on Digital UNIX. ♦

**DICTIONARY**
OpenVMS VAX OpenVMS Alpha Displays the current default dictionary directory in the data dictionary. This display is disabled on Digital UNIX. ♦

**DIGIT SEPARATOR**
Displays the character that will be used as the digit separator in output displays. (The digit separator is the symbol that separates groups of three digits in values greater than 999. For example, the comma is the digit separator in the number 1,000.)

**EXECUTION MODE**
Shows whether or not SQL executes the statements that you issue in your interactive SQL session. The default is to execute the statements as you issue them. However, if you have issued a SET NOEXECUTE MODE statement in your session, SQL will not execute subsequent statements.

You can use the SET NOEXECUTE MODE statement to display access strategies and check for syntax errors. For more information, see the SET Statement.

**FLAGGER MODE**
Shows whether or not SQL flags statements containing nonstandard syntax for all set flaggers. If you specify SET FLAGGER ON, which is equivalent to SET FLAGGER SQL92_ENTRY ON, the SHOW FLAGGER statement informs you that flagging for the ANSI/ISO standard is set. If you specified SET FLAGGER MIA ON, the SHOW FLAGGER statement informs you that flagging for the MIA standard is set.

### LANGUAGE

OpenVMS VAX OpenVMS Alpha Displays the language to be used for translation of month names and abbreviations in date and time input and display. The language name also determines the translation of other language-dependent text, such as the translation for the date literals YESTERDAY, TODAY, and TOMORROW. This display is disabled on Digital UNIX. ♦

### RADIX POINT

Displays the character that will be used as the radix point in output displays. (The radix point is the symbol that separates units from decimal fractions. For example, in the number 98.6, the period is the radix point.)

### SQLCA

Displays the contents of the SQL Communications Area (SQLCA). The SQLCA is a collection of variables that SQL uses to provide information about the execution of SQL statements to application programs. In interactive SQL, you can use the SHOW SQLCA statement to learn about the different variables in the SQLCA. See Appendix B for more information about the SQLCA.

### TRANSACTION

Displays the characteristics of the current transaction or, if there is no active transaction, the characteristics specified in the last DECLARE TRANSACTION statement. For each database within the scope of the transaction, SQL displays the following:

- Transaction
- Tables specified in the RESERVING clause of the DECLARE TRANSACTION or SET TRANSACTION statement
- Share mode and lock type for each of those tables
- If fast commit processing is enabled

In addition, the SHOW TRANSACTION statement displays transaction information returned by the base database system about the transaction, such as whether or not the transaction is active.

### VERSIONS

Displays the version of SQL and the underlying software components.

### WARNING MODE

Displays the default setting for warning messages. If WARNING MODE is set to ON, SQL flags statements containing obsolete SQL syntax. Obsolete syntax is syntax that was allowed in previous versions of SQL but has changed. Oracle Rdb recommends that you avoid using such syntax because it may not be supported in future versions. By default, SQL displays a warning message after any statement containing obsolete syntax (WARNING MODE ON).

To suppress messages about obsolete syntax, use the SET WARNING NODEPRECATE statement.

**STORAGE AREAS**
Displays information about storage areas. If you do not specify a wildcard or list of storage area names, SQL displays the names of all the storage areas in all attached databases.

**USAGE**
Displays the usage, object name, storage map, and storage map partition number for the specified storage area. Partition numbers are always shown in parentheses, (1), and may be accompanied by a storage map name. For example, for an index there is no special map because it is part of the index. For a table, the map is an extra object and therefore is reported. For example:

```
SQL> SHOW STORAGE AREA (USAGE) EMPIDS_LOW

Database objects using Storage Area EMPIDS_LOW:
Usage            Object Name                     Map / Partition
---------------- ------------------------------- ----------------------------
Index            EMPLOYEES_HASH                  (1)
Storage Map      EMPLOYEES                       EMPLOYEES_MAP (1)
Index            JOB_HISTORY_HASH                (1)
Storage Map      JOB_HISTORY                     JOB_HISTORY_MAP (1)
```

**ATTRIBUTES**
Displays storage area type, access, page format, page size, storage area file, storage area allocation, storage area extent minimum and maximum, storage area extent percent, snapshot file, snapshot allocation, snapshot extent minimum and maximum, snapshot extent percent, whether extents are enabled or disabled, and the locking level for the specified storage area.

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. In a SHOW STORAGE AREAS statement, specifying a wildcard displays storage area type, access, page format, page size, storage area file, storage area allocation, storage area extent minimum and maximum, storage area extent percent, snapshot file, snapshot allocation, snapshot extent minimum and maximum, snapshot extent percent, whether extents are enabled or disabled, and the lock level for storage areas. If you do not precede the wildcard with an alias, SQL displays information about storage areas for the default database. If you precede the

wildcard with an alias, SQL displays information about storage areas in that database.

**storage-area-name**
Specifies the name of the storage area whose information you want to display. SQL displays storage area type, access, page format, page size, storage area file, storage area allocation, storage area extent minimum and maximum, storage area extent percent, snapshot file, snapshot allocation, snapshot extent minimum and maximum, snapshot extent percent, whether extents are enabled or disabled, and the lock level for each storage area you specify.

**STORAGE MAPS**
Displays information about storage maps. If you do not specify a wildcard or list of storage map names, SQL displays the names of all the storage maps in all attached databases.

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. In a SHOW STORAGE MAPS statement, specifying a wildcard displays whether compression is enabled or disabled, and the store clause from the CREATE STORAGE MAP statement for storage maps. If you do not precede the wildcard with an alias, SQL displays information about storage maps for the default database. If you precede the wildcard with an alias, SQL displays information about storage maps in that database.

**storage-map-name**
Specifies the name of the storage map whose information you want to display. SQL displays whether compression is enabled or disabled and the store clause from the CREATE STORAGE MAP statement for each storage map you specify.

**TABLES**
Displays information about tables and views. If you do not specify a wildcard or list of table and view names, SQL displays the names of all the tables and views in all attached databases.

If you do not specify any of the SHOW TABLES options (COLUMNS, COMMENT, CONSTRAINTS, INDEXES, STORAGE MAPS, or TRIGGERS), by default you will see the display for all these options including the character set for each column of the specified table.

**COLUMNS**
Displays each column name, data type, and domain name for the specified tables.

**COMMENT**
Displays the comments for the specified tables.

**CONSTRAINTS**
Displays the constraints for the specified tables and the constraints referencing the specified tables. The display shows the name and type of each constraint, its evaluation time, and its source definition.

**INDEXES**
Displays the indexes for the specified tables. The display shows the name and type of each index, if duplicates are allowed, and if compression is enabled or disabled.

**STORAGE MAPS**
Displays the names of the storage maps for the specified tables.

**TRIGGERS**
Displays the names and source definitions of triggers for the specified tables.

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. In a SHOW TABLES statement, specifying a wildcard displays the column definitions, constraints, storage maps, and indexes for tables, and the column definitions and select expressions for views. If you do not precede the wildcard with an alias, SQL displays information about tables and views for the default database. If you precede the wildcard with an alias, SQL displays information about tables and views in that database.

**table-name**
Specifies the name of the table or view whose definition you want to display. SQL displays the name, column definitions, constraints, storage maps, and indexes for each table you specify.

**TRIGGERS**
Displays information about triggers. If you do not specify a wildcard or a trigger name, SQL displays the names of all the triggers in all attached databases.

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. In a SHOW TRIGGERS statement, specifying a wildcard displays the names and definitions of triggers. If you do not precede the wildcard with an alias,

SQL displays information about triggers for the default database. If you precede the wildcard with an alias, SQL displays information about triggers in that database.

**trigger-name**
Specifies the name of the trigger whose definition you want to display. SQL displays the name and definition of the trigger or triggers you specify.

**SHOW USERS GRANTING**
Displays all the users who gave a particular privilege to a particular user. This statement displays the privileges that need to be revoked to take a privilege away from the user, either directly or indirectly.

**ON TABLE table-name**
**ON COLUMN column-name**
**ON DATABASE alias**
**ON FUNCTION ext-function-name**
**ON PROCEDURE ext-procedure-name**
**ON MODULE module-name**
Specifies the tables, columns, databases, external functions, external procedures, or modules in an ANSI/ISO-style database for which you want to display users granting privileges. You can specify a list of tables, columns, databases, external functions, external procedures, or modules, but you must specify at least one item to display a list.

**TO identifier-ansi-style**
**TO PUBLIC**
Specifies the identifiers for the new or modified access privilege set entry. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

**SHOW USERS WITH**
Displays all the users who received a particular privilege from a particular user, including all the users who indirectly received privileges. This is also the list of users who lose a particular privilege when it is taken away from any users who granted the privilege.

**ON TABLE table-name**
**ON COLUMN column-name**
**ON DATABASE alias**
**ON FUNCTION ext-function-name**
**ON PROCEDURE ext-procedure-name**
**ON MODULE module-name**
Specifies the tables, columns, databases, external functions, external procedures, or modules in an ANSI/ISO-style database for which you want to display users receiving privileges. You can specify a list of tables, columns, databases, external functions, external procedures, or modules, but you must specify at least one item to display a list.

**FROM identifier-ansi-style**
**FROM PUBLIC**
Specifies the identifiers for the new or modified access privilege set entry. Specifying PUBLIC is equivalent to a wildcard specification of all user identifiers.

**VARIABLES**
Displays information about declared variables.

**VIEWS**
Displays information about views. If you do not specify a wildcard or list of view names, SQL displays the names of all the views in all attached databases.

If you do not specify any of the SHOW VIEW options (COLUMNS, COMMENT, or SOURCE), by default you will see the display for all these options.

**COLUMNS**
Displays each column name, data type, and domain name for the specified views.

**COMMENT**
Displays the comments for the specified views.

**SOURCE**
Displays the source definitions for the specified views.

*****
**alias.***
Specifies an asterisk wildcard, preceded by an optional alias. In a SHOW VIEWS statement, specifying a wildcard displays the column definitions and select expressions for views. If you do not precede the wildcard with an alias, SQL displays information about views in the default database. If you precede

the wildcard with an alias, SQL displays information about views in that database.

**view-name**
Specifies the name of the view whose definition you want to display. SQL displays the name and column definitions for each view you specify.

## Usage Notes

- If the database default character set and the national character set for the database differ from the session character sets, the SHOW ALIASES and SHOW DATABASES statements display the character sets for the specified database.
- If the character set of a domain or table is different than the database default character set, the SHOW DOMAINS and SHOW TABLES statements display the character set of the specified domain or table. Otherwise, the display of the character set information is suppressed.
- The SHOW INDEXES statement displays the size of the key for the specified index.
- The SHOW TABLES statement displays the character set and the length of each column.
- If you attach to the same database twice, SHOW statements may fail with a deadlock error. You can avoid this error by issuing a COMMIT statement.

```
SQL> ATTACH 'FILENAME CORPORATE_DATA';
SQL> ATTACH 'ALIAS CORP2 FILENAME CORPORATE_DATA';
SQL> SHOW TABLES;
User tables in database with filename corporate_data
     DAILY_HOURS
     DEPARTMENTS
     PAYROLL
     .
     .
     .
     PERSONNEL.WEEKLY_WAGES          A view.
     RECRUITING.CANDIDATES
     RECRUITING.COLLEGES
     RECRUITING.DEGREES
     RECRUITING.RESUMES
```

```
User tables in database with alias CORP2
%RDB-F-DEADLOCK, request failed due to resource deadlock
-RDMS-F-DEADLOCK, deadlock on record 41:413:1
SQL> COMMIT;
SQL> SHOW TABLES;
     User tables in database with filename corporate_data
     DAILY_HOURS
     DEPARTMENTS
     PAYROLL
     .
     .
     .
     User tables in database with alias CORP2
     "CORP2.ADMINISTRATION".ACCOUNTING.DAILY_HOURS
     "CORP2.ADMINISTRATION".ACCOUNTING.DEPARTMENTS
     "CORP2.ADMINISTRATION".ACCOUNTING.PAYROLL
     .
     .
     .
     "CORP2.ADMINISTRATION".RECRUITING.COLLEGES
     "CORP2.ADMINISTRATION".RECRUITING.DEGREES
     "CORP2.ADMINISTRATION".RECRUITING.RESUMES
```

- The definitions appearing in the SHOW examples in the following sections are in SQL format because it is assumed that SQL was the source language used to create the definitions.

## Examples

Example 1: Using the SHOW statement displays

The following log file from an interactive SQL session illustrates some of the arguments for the SHOW statement:

```
SQL> -- Show the session character sets.
SQL> --
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
SQL> --
SQL> -- Attach to the database and show database character sets.
SQL> --
SQL> ATTACH 'FILENAME MIA_CHAR_SET';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

```
Alias RDB$DBHANDLE:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI

SQL> --
SQL> -- Attach to the second database and show character sets of both.
SQL> --
SQL> ATTACH 'ALIAS MIA1 FILENAME MIA_CHAR_SET';
SQL> SHOW CHARACTER SETS;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS

Alias RDB$DBHANDLE:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
SQL> --
SQL> -- SHOW ALIASES examples.
SQL> --
SQL> SHOW ALIASES;
Default alias:
    Rdb database in file MIA_CHAR_SET
Alias MIA1:
    Rdb database in file MIA_CHAR_SET
SQL> SHOW ALIAS MIA1;
Alias MIA1:
    Rdb database in file MIA_CHAR_SET
        Multischema mode is disabled
        Default character set is DEC_KANJI
        National character set is KANJI
        Identifier character set is DEC_KANJI
        Number of users:              50
        Number of nodes:              16
   .
   .
   .
        ACL based protections
Storage Areas in database with alias MIA1
     RDB$SYSTEM                      List storage area.
```

```
SQL> --
SQL> -- SHOW CONNECTIONS examples.
SQL> --
SQL> CONNECT TO 'ALIAS MIA1 FILENAME MIA_CHAR_SET' AS 'TEST';
SQL> SHOW CONNECTIONS;
        -default-
->      TEST
SQL> SHOW CONNECTIONS DEFAULT;
Connection: -default-
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
   .
   .
   .
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS

Alias RDB$DBHANDLE:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI

SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: SQLV40
Default DATE type: DATE VMS
Quoting Rules: SQLV40
Optimization Level: DEFAULT
Hold Cursors default: WITH HOLD PRESERVE NONE
Quiet commit mode: OFF
Compound transactions mode: EXTERNAL
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
```

```
Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI

SQL> --
SQL> CONNECT TO 'ALIAS MIA1 FILENAME MIA_CHAR_SET' AS 'test1';
SQL> --
SQL> -- You must set quoting rules to the SQL92 environment and use
SQL> -- double quotation marks (") to display the settings of the
SQL> -- 'test1' connection or use SHOW CONNECTIONS CURRENT.
SQL> --
SQL> SHOW CONNECTIONS;
        -default-
        TEST
->      test1
SQL> SHOW CONNECTIONS test1;
Connection: TEST1
%SQL-F-NOSUCHCON, There is not an active connection by that name
SQL> SET QUOTING RULES 'SQL92';
SQL> SHOW CONNECTIONS "test1";
Connection: test1
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is DAY
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: SQLV40
Default DATE type: DATE VMS
Quoting Rules: ANSI/ISO
Optimization Level: DEFAULT
Hold Cursors default: WITH HOLD PRESERVE NONE
Quiet commit mode: OFF
Compound transactions mode: EXTERNAL
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is DEC_KANJI
        National character set is KANJI
```

```
SQL> SET CONNECT DEFAULT;
SQL> --
SQL> -- SHOW DATABASES examples.
SQL> --
SQL> SHOW DATABASES;
Default alias:
    Oracle Rdb database in file MIA_CHAR_SET
Alias MIA1:
    Oracle Rdb database in file MIA_CHAR_SET
SQL> SHOW DATABASE RDB$DBHANDLE;
Default alias:
    Oracle Rdb database in file MIA_CHAR_SET
        Multischema mode is disabled
        Default character set is DEC_KANJI
        National character set is KANJI
        Identifier character set is DEC_KANJI
        Number of users:               50
   .
   .
   .
SQL> --
SQL> -- SHOW DOMAINS example.
SQL> --
SQL> SHOW DOMAINS;
User domains in database with filename MIA_CHAR_SET
ARABIC_DOM                      CHAR(8)
         ISOLATINARABIC 8 Characters,  8 Octets
DEC_KANJI_DOM                   CHAR(16)
GREEK_DOM                       CHAR(8)
         ISOLATINGREEK 8 Characters,  8 Octets
HINDI_DOM                       CHAR(8)
         DEVANAGARI 8 Characters,  8 Octets
KANJI_DOM                       CHAR(8)
         KANJI 4 Characters,  8 Octets
KATAKANA_DOM                    CHAR(8)
         KATAKANA 8 Characters,  8 Octets
MCS_DOM                         CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
RUSSIAN_DOM                     CHAR(8)
         ISOLATINCYRILLIC 8 Characters,  8 Octets

SQL> --
SQL> -- SHOW TABLES example.
SQL> --
SQL> SHOW TABLES;
User tables in database with filename MIA_CHAR_SET
     COLOURS
SQL> SHOW TABLE (COLUMNS) COLOURS;
Information for table COLOURS
```

```
Columns for table COLOURS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
ENGLISH                         CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
FRENCH                          CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
JAPANESE                        CHAR(8)          KANJI_DOM
         KANJI 4 Characters,  8 Octets
ROMAJI                          CHAR(16)         DEC_KANJI_DOM
KATAKANA                        CHAR(8)          KATAKANA_DOM
         KATAKANA 8 Characters,  8 Octets
HINDI                           CHAR(8)          HINDI_DOM
         DEVANAGARI 8 Characters,  8 Octets
GREEK                           CHAR(8)          GREEK_DOM
         ISOLATINGREEK 8 Characters,  8 Octets
ARABIC                          CHAR(8)          ARABIC_DOM
         ISOLATINARABIC 8 Characters,  8 Octets
RUSSIAN                         CHAR(8)          RUSSIAN_DOM
         ISOLATINCYRILLIC 8 Characters,  8 Octets

SQL> --
SQL> -- SHOW INDEXES example.
SQL> --
SQL> SHOW INDEXES;
User indexes in database with filename MIA_CHAR_SET
     COLOUR_INDEX
SQL> SHOW INDEXES COLOUR_INDEX;
Indexes on table COLOURS:
COLOUR_INDEX                    with column JAPANESE
                                size of index key is 8 octets
  Duplicates are allowed
  Type is Sorted
```

Example 2: Showing features that internationalize your terminal session

The following example displays SHOW statements that let you see the values for the SET statements dealing with internationalization:

```
SQL> --
SQL> -- First, use the SET statement to specify nondefault values.
SQL> --
SQL> SET CURRENCY SIGN '£'
SQL> --
SQL> SET DATE FORMAT TIME 15
SQL> --
SQL> SET DIGIT SEPARATOR '.'
SQL> --
SQL> SET LANGUAGE GERMAN
SQL> --
SQL> SET RADIX POINT ','
SQL> --
SQL> -- Now look at the SHOW displays.
SQL> --
SQL> SHOW CURRENCY SIGN
Currency sign is '£'.
SQL> --
SQL> SHOW DATE FORMAT
Date format is TIME 15.
SQL> --
SQL> SHOW DIGIT SEPARATOR
Digit separator is '.'.
SQL> --
SQL> SHOW LANGUAGE
Language is GERMAN.
SQL> --
SQL> SHOW RADIX POINT
Radix point is ','.
```

Example 3: Showing the setting for nonstandard syntax flagging

```
SQL> SHOW FLAGGER MODE
The flagger mode is OFF
SQL> SET FLAGGER SQL92_ENTRY ON
SQL> SHOW FLAGGER MODE
%SQL-I-NONSTASYN92E, Nonstandard SQL92 Entry-level syntax
The SQL92 Entry-level flagger mode is ON
```

Example 4: Showing after-image journal files

The following example displays journal information:

```
SQL> ATTACH 'FILENAME SAMPLE';
SQL> SHOW JOURNAL * ;
Journals in database with filename SAMPLE
     AIJ_ONE
        Journal File:    DISK1:[DIRECTORY1]AIJ_ONE.AIJ;1
        Backup File:     DISK2:[DIRECTORY2]AIJ_ONE.AIJ;
 Edit String:     ()
     AIJ_TWO
        Journal File:    DISK1:[DIRECTORY1]AIJ_TWO.AIJ;1
        Backup File:     DISK2:[DIRECTORY2]AIJ_TWO.AIJ;
        Edit String:     ('$'+HOUR+MINUTE+'_'+MONTH+DAY+'_'+SEQUENCE)
```

Example 5: Showing storage area usage and attribute information

The following example displays storage area information:

```
SQL> -- Display the usage of storage area TEST_AREA and JOBS
SQL> --
SQL> SHOW STORAGE AREAS (USAGE) TEST_AREA
No database objects use Storage Area TEST_AREA

SQL> SHOW STORAGE AREAS (USAGE) JOBS
```

```
Database objects using Storage Area JOBS:
Usage            Object Name                     Map / Partition
---------------- ------------------------------- -----------------------
Storage Map      JOBS                            JOBS_MAP (1)
SQL> --
SQL> -- Display the attributes of storage area JOBS.
SQL> --
SQL> SHOW STORAGE AREAS (ATTRIBUTES) JOBS

     JOBS
         Access is:      Read write
         Page Format:    Mixed
         Page Size:      2 blocks
         Area File:      SQL_USER1:[DAY.V60]JOBS.RDA;1
         Area Allocation:          27 pages
         Area Extent Minimum:      99 pages
         Area Extent Maximum:      9999 pages
         Area Extent Percent:      20 percent
         Snapshot File:  SQL_USER1:[DAY.V60]JOBS.SNP;1
         Snapshot Allocation:      10 pages
         Snapshot Extent Minimum:  99 pages
         Snapshot Extent Maximum:  9999 pages
         Snapshot Extent Percent:  20 percent
         Extent :       Enabled
         Locking is Row Level
  Using Cache CACHE1
```

Example 6: Showing query outline information

The following example displays query outline information:

```
SQL> SHOW OUTLINE MY_OUTLINE
     MY_OUTLINE
 Source:

create outline MY_OUTLINE
id '09ADFE9073AB383CAABC4567BDEF3832'
mode 0
as (
  query (
    subquery (
      EMPLOYEES 0     access path index       EMP_LAST_NAME
        join by cross to
      DEGREES 1       access path index       DEG_EMP_ID
      )
    )
  )
compliance optional     ;
```

Example 7: Showing privileges

The following example demonstrates the SHOW PRIVILEGES statement:

```
SQL> ! Attach as the logged on user, [sql,heleng]
SQL>  ATTACH 'FILENAME personnel';
SQL> SHOW PRIVILEGES ON DATABASE RDB$DBHANDLE
Privileges on Alias RDB$DBHANDLE
    (IDENTIFIER=[sql,heleng],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
SQL> !
SQL> ! Attach as user rhonda.
SQL>  ATTACH 'FILENAME personnel USER ''rhonda'' USING ''newhampshire''';
SQL> ! User rhonda has SELECT privilege.
SQL> SHOW PRIVILEGES ON DATABASE RDB$DBHANDLE
Privileges on Alias RDB$DBHANDLE
    (IDENTIFIER=[sql,rhonda],ACCESS=SELECT)
SQL> EXIT
$ !
$ ! On OpenVMS, give the process the BYPASS privilege, which
$ ! gives you access to any database object.
$ SET PROC/PRIVILEGES=BYPASS
$ SQL$
SQL> ! Attach as user rhonda.
SQL>  ATTACH 'FILENAME personnel USER ''rhonda'' USING ''newhampshire''';
SQL> !
SQL> ! User rhonda now has all privileges, inherited from the logged-on
SQL> ! process.
SQL> SHOW PRIVILEGES ON DATABASE RDB$DBHANDLE
Privileges on Alias RDB$DBHANDLE
    (IDENTIFIER=[sql,rhonda],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
```

Example 8: Showing modules, stored procedures, and stored functions

```
SQL> --
SQL> -- Show the modules in the database.
SQL> --
SQL> ATTACH 'FILENAME mf_personnel';
SQL> SHOW MODULES
Modules in database with filename mf_personnel
 Module name is: UTILITY_FUNCTIONS
```

```
SQL> SHOW MODULES utility_functions
 Module name is: UTILITY_FUNCTIONS
 Header:
 utility_functions
                     language sql
 No description found.
 Owner is:
 Module ID is: 1
Functions/Procedures in Module:
        Function  ABS
        Function  MDY
        Procedure TRACE_DATE

SQL> --
SQL> -- Show the procedures and functions of the module.
SQL> --
SQL> SHOW MODULES (PROCEDURES) utility_functions
 Module name is: UTILITY_FUNCTIONS
Functions/Procedures in Module:
        Function  ABS
        Function  MDY
        Procedure TRACE_DATE

SQL> SHOW PROCEDURE trace_date
Procedure name is: TRACE_DATE
 Procedure ID is: 3
 Source:
 trace_date (:dt date);
                     begin
                     trace :dt;
                     end
 No description found.
 Module name is: UTILITY_FUNCTIONS
 Module ID is: 1
 Number of parameters is: 1

Parameter Name                Data Type
--------------                ---------

DT                            DATE VMS
        Parameter position is 1
        Parameter is IN (read)
        Parameter is passed by REFERENCE
```

```
SQL> SHOW FUNCTIONS abs
Function name is: ABS
 Function ID is: 2
 Source:
 abs (in :arg integer) returns integer
                    comment 'Returns the absolute value of an integer';
                    begin
                    return case
                    when :arg < 0 then - :arg
                    else :arg
                    end;
                    end
 Comment:       Returns the absolute value of an integer
 Module name is: UTILITY_FUNCTIONS
 Module ID is: 1
 Number of parameters is: 1

Parameter Name                 Data Type
--------------                 ---------

                               INTEGER
        Function result datatype
        Return value is passed by VALUE

ARG                            INTEGER
        Parameter position is 1
        Parameter is IN (read)
        Parameter is passed by REFERENCE
```

Example 9: Showing a storage map that defines both horizontal and vertical record partitioning

```
SQL> SHOW STORAGE MAP  EMPLOYEES_1_MAP2
     EMPLOYEES_1_MAP2
 For Table:             EMP2
 Partitioning is:       UPDATABLE
 Store clause:          STORE COLUMNS (EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
                        MIDDLE_INITIAL, STATUS_CODE)
            USING (EMPLOYEE_ID)
               IN ACTIVE_AREA_A WITH LIMIT OF ('00399')
               IN ACTIVE_AREA_B WITH LIMIT OF ('00699')
               OTHERWISE IN ACTIVE_AREA_C
         STORE COLUMNS (ADDRESS_DATA_1, ADDRESS_DATA_2, CITY,
                        STATE, POSTAL_CODE)
            USING (EMPLOYEE_ID)
               IN INACTIVE_AREA_A WITH LIMIT OF ('00399')
               IN INACTIVE_AREA_B WITH LIMIT OF ('00699')
               OTHERWISE IN INACTIVE_AREA_C
         STORE IN OTHER_AREA
 Compression is:        ENABLED
 Partition 2:           Compression is Enabled
 Partition 3:           Compression is Enabled
```

## Simple Statement

Includes a single SQL statement in a module procedure or in an embedded host language program. The statement can include a single executable SQL statement. A module procedure or embedded procedure that contains a simple statement is called a **simple-statement procedure**.

Table 1-1 lists all the SQL statements allowed in a simple statement.

### Environment

A simple statement is valid either in a procedure of an SQL module file or in an embedded host language program prefixed by the keywords EXEC SQL:

- Module SQL

  See Section 3.2 for information about using simple statements in module procedures in an SQL module file.

- Embedded SQL

  See Section 4.2 for information about using simple statements in embedded procedures in host language programs.

### Format

simple-statement =

SQL statement

### Arguments

**SQL statement**
Specifies a single executable SQL statement.

Executable SQL statements undergo processing during module compile time but do not execute until the program runs. SQL executes the simple statement when the procedure in which it is embedded is called by a host language module. (Nonexecutable SQL statements are those that SQL processes completely when it compiles an SQL module but are not executed at run time.) See Section 1.5 for information about which SQL statements are executable.

The SQL statement must use names specified in the procedure's formal parameters wherever it refers to parameters.

## Usage Notes

- A simple statement can contain only one SQL statement for each procedure; however, you can include more than one statement in a procedure if you specify a compound statement. (A module or embedded procedure that contains a compound statement is called a **multistatement procedure**.) Currently, SQL imposes fewer restrictions on simple-statement procedures than on multistatement procedures, but multistatement procedures execute more efficiently. Oracle Rdb suggests that you use multistatement procedures wherever possible. See the Compound Statement for more information.
- If the statement is contained within a procedure, it must end with a semicolon.

## Examples

Example 1: A simple statement using interactive SQL

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont> JOURNAL IS DISABLED;
```

## SIGNAL Control Statement

Passes the signaled SQLSTATE status parameter back to the application or SQL interface and terminates the current routine and all calling routines.

### Environment

You can use the SIGNAL statement in a compound statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
SIGNAL ──► char-value-expr ──►
```

### Arguments

**SIGNAL char-value-expr**
Expects one character value expression which is used as the SQLSTATE status parameter. Any provided value expression is converted to a CHAR(5) value which is passed to SIGNAL.

See Section 2.6 for more information on value expressions. See Appendix C for more information about SQLSTATE.

### Usage Notes

- The current routine and all calling routines are terminated and the signaled SQLSTATE status parameter is passed to the application.
- The SQLSTATE value is mapped to the SQLCODE status parameter.
- If the SQLSTATE status parameter value maps to more than one SQLCODE value, the SQLCODE is set to the value -1042.

- The contents of the SQLSTATE status parameter string are defined by the ANSI/ISO SQL Standard and must contain only Latin capital letters (A through Z) or Arabic digits (0 through 9). Any string longer than 5 characters is truncated. Any string shorter than 5 characters is space-filled which causes an error to be returned. The character set for the string must be ASCII or DEC_MCS.
- If a numeric value expression is used with SIGNAL or is assigned to a character variable used with SIGNAL, the value is converted to a character string with possible leading spaces. The leading spaces are considered invalid. For example, SIGNAL 02000 is considered invalid, but SIGNAL '02000' is acceptable.
- If the SQLSTATE string contains invalid characters, Oracle Rdb generates the following error:

```
%RDB-F-CONVERT_ERROR, invalid or unsupported data conversion
-RDMS-E-SQLSTATE_ILLCH, illegal character in SQLSTATE string passed to
SIGNAL routine
```

- If the character value expression results in a null value, Oracle Rdb generates the following error:

```
%RDB-F-CONVERT_ERROR, invalid or unsupported data conversion
-RDMS-E-SQLSTATE_NULL, unexpected NULL passed to SIGNAL routine
```

- When Oracle Rdb encounters an error, the error message returned by Oracle Rdb includes the name of the routine that returned the error. If the routine is an unnamed compound statement or multistatement procedure, the error message specifies "(unnamed)". For example:

```
%RDB-F-SIGNAL_SQLSTATE, routine "(unnamed)" signaled SQLSTATE "22028"
```

- SQL applications can examine the SQLSTATE variable to see what was signaled by SQL or an application SIGNAL call.

## Examples

Example 1: Using the SIGNAL and RETURN statements, multiline comments, and stored functions

Application programmers are often faced with the problem of generating unique sequence numbers for use as primary keys. This example shows one solution, using SQL stored functions to maintain a unique sequence-numbered table, and issue values when called.

The example uses a table, NEXT_KEY_TABLE, to maintain a list of key names and their current values. In this example, only a single key is created with the name EMPLOYEE_ID. Each time the function is called, it fetches the value from the NEXT_KEY_TABLE and returns the next value. If the named key is not found, an error is returned (SQLSTATE 22023 is defined as "invalid parameter value").

Other solutions include executing an AFTER INSERT trigger, which calculates the next sequence number and stores it as part of the primary key. The main advantage of the SQL stored function solution is that the final value is inserted and no subsequent update is required to the data row. This saves unnecessary validation queries for the primary key field.

Some other enhancements that could be made to this example include:

- Defining an index on the NEXT_KEY_TABLE to improve retrieval of the key value
- Adding parameters to the function to allow the current value to be returned (rather than always returning the next value)

```
SQL> CREATE DOMAIN key_name
cont>   CHAR(31)
cont>   CHECK (VALUE IS NOT NULL)
cont>   NOT DEFERRABLE;
SQL> --
SQL> CREATE TABLE next_key_table (
cont>   next_key_val INTEGER NOT NULL,
cont>   next_key_name key_name UNIQUE);
SQL> --
SQL> INSERT INTO next_key_table (next_key_name, next_key_val)
cont>   VALUES ('EMPLOYEE_ID', 0);
1 row inserted
SQL> --
SQL> CREATE MODULE tools
cont>   LANGUAGE SQL
cont>   FUNCTION next_key (IN :key_name key_name)
cont>      RETURNS INTEGER
cont>      COMMENT IS 'This routine fetches the next value of the'/
cont>                 'specified entry in the sequence table.  The'/
cont>                 'passed name is converted to uppercase before'/
cont>                 'retrieval (see the DEFAULT clause for compound'/
cont>                 'statements).  The UPDATE ... RETURNING statement'/
cont>                 'is used to fetch the new value after the update.'/
cont>                 'If no entry exists, then an error is returned.';
cont>      BEGIN
cont>        DECLARE :rc, :new_val INTEGER DEFAULT 0;
cont>        DECLARE :key_name_upper key_name DEFAULT UPPER(:key_name);
cont>        DECLARE :invalid_parameter CONSTANT CHAR(5) = '22023';
cont> --
cont>        UPDATE next_key_table
cont>        SET next_key_val = next_key_val + 1
cont>        WHERE next_key_name = :key_name_upper
cont>        RETURNING next_key_val
cont>        INTO :new_val;
cont> --
cont>        GET DIAGNOSTICS :rc = ROW_COUNT;
cont>        TRACE 'NEXT_KEY is ', COALESCE(:new_val, 'NULL'), ', RC is ', :rc;
cont> --
cont>        IF :rc = 0 THEN
cont>            TRACE 'No entry exists for KEY_NAME: ', :key_name_upper;
cont>            SIGNAL :invalid_parameter;
cont>         ELSE
cont>            TRACE 'Returning new value for ', :key_name_upper, :new_val;
cont>            RETURN :new_val;
cont>        END IF;
cont> --
cont>      END;
cont> END MODULE;
SQL> --
SQL> CREATE TABLE employee (
cont>   employee_id         INTEGER,
cont>   last_name           CHAR(20),
cont>   birthday            DATE);
SQL> --
SQL> -- Turn on the TRACE flag so we can see the function working.
SQL> --
SQL> SET FLAGS 'TRACE';
SQL> --
SQL> INSERT INTO employee (employee_id, last_name, birthday)
cont>   VALUES (next_key('EMPLOYEE_ID'), 'Smith', DATE'1970-1-1');
~Xt: NEXT_KEY is 1          , RC is 1
~Xt: Returning new value for EMPLOYEE_ID                   1
1 row inserted
SQL> --
SQL> INSERT INTO employee (employee_id, last_name, birthday)
cont>   VALUES (next_key('EMPLOYEE_ID'), 'Lee', DATE'1971-1-1');
~Xt: NEXT_KEY is 2          , RC is 1
~Xt: Returning new value for EMPLOYEE_ID                   2
1 row inserted
SQL> --
SQL> INSERT INTO employee (employee_id, last_name, birthday)
cont>   VALUES (next_key('EMPLOYEE_ID'), 'Zonder', DATE'1972-1-1');
~Xt: NEXT_KEY is 3          , RC is 1
~Xt: Returning new value for EMPLOYEE_ID                   3
1 row inserted
```

```
SQL> --
SQL> SELECT * FROM employee ORDER BY EMPLOYEE_ID;
 EMPLOYEE_ID   LAST_NAME              BIRTHDAY
           1   Smith                  1970-01-01
           2   Lee                    1971-01-01
           3   Zonder                 1972-01-01
3 rows selected
SQL> --
SQL> -- Show the error if the unknown key_name is passed.
SQL> --
SQL> INSERT INTO employee (employee_id, last_name, birthday)
cont>   VALUES (next_key('EMPLOYEEID'), 'Zonder', DATE'1972-1-1');
~Xt: NEXT_KEY is 0          , RC is 0
~Xt: No entry exists for KEY_NAME: EMPLOYEEID
%RDB-E-SIGNAL_SQLSTATE, routine "NEXT_KEY" signaled SQLSTATE "22023"
```

## TRACE Control Statement

Writes values to the trace log file after the trace extended debug flag is set. The TRACE control statement lets you specify multiple value expressions. It stores a value in a log file for each value expression it evaluates.

SQL turns on trace logging only if the logical name RDMS$DEBUG_FLAGS or configuration parameter RDB_DEBUG_FLAGS is defined to be *Xt*. The letter *X* must be an uppercase letter and the letter *t* must be in lowercase.

Trace logging can help you debug complex multistatement procedures.

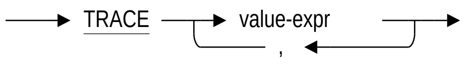
### Environment

You can use the TRACE control statement in a compound statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

trace-statement =



### Arguments

**value-expr**
Specifies a symbol or string of symbols used to represent or calculate a single value.

See Section 2.6 for a complete description of the variety of value expressions that SQL provides.

## Usage Notes

- The TRACE control statement has no effect when the debug flag is undefined.

  Output can be redirected using the RDMS$DEBUG_FLAGS_OUTPUT logical name or RDB_DEBUG_FLAGS_OUTPUT configuration parameter. See Appendix E and the *Oracle Rdb7 Guide to Database Performance and Tuning* for information on logical names and configuration parameters.
- NULL is displayed if a value expression is the null value.
- The trace feature can also be enabled by specifying the SET FLAGS 'TRACE' statement. See the SET FLAGS Statement for more information.
- You can trace IN, OUT, and INOUT parameters. For example:

```
SQL> CREATE MODULE m1
cont> LANGUAGE SQL
cont>    PROCEDURE p1 (IN :a INTEGER, OUT :b REAL);
cont>       BEGIN
cont>          SET :b = :a;
cont>          TRACE :a, :b;
cont>       END;
cont> END MODULE;
SQL> SET FLAGS 'TRACE';
SQL> DECLARE :res real;
SQL> CALL p1 (10, :res);
~Xt: 10         1.0000000E+01
          RES
  1.0000000E+01
```

## Examples

Example 1: Tracing a multistatement procedure

```
$ DEFINE RDMS$DEBUG_FLAGS "Xt"
$ SQL
SQL> ATTACH 'FILENAME MF_PERSONNEL';
SQL> DECLARE :i INTEGER;
SQL> BEGIN
cont>    WHILE :i <= 10
cont>     LOOP
cont>         TRACE ':i is', :i;
cont>         SET :i = :i +1;
cont>     END LOOP;
cont> END;
~Xt: :i is 0
~Xt: :i is 1
~Xt: :i is 2
~Xt: :i is 3
~Xt: :i is 4
~Xt: :i is 5
~Xt: :i is 6
~Xt: :i is 7
~Xt: :i is 8
~Xt: :i is 9
~Xt: :i is 10
```

## TRUNCATE TABLE Statement

Deletes the data in a table while still maintaining the metadata definitions of the table. The TRUNCATE TABLE statement is equivalent to the DROP TABLE CASCADE statement followed by a CREATE TABLE statement including the reconstruction of the target and referencing metadata.

### Environment

You can use the TRUNCATE TABLE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

### Format

```
TRUNCATE TABLE  ───►  <table-name>    ───►
```

### Arguments

**table-name**
Specifies the name of the table you want to truncate.

### Usage Notes

- The TRUNCATE TABLE statement resets:
  - All Indexes
  - Any storage maps on the table
- The TRUNCATE TABLE statement does not:
  - Execute any delete triggers
  - Invalidate procedures
  - Invalidate query outlines and stored routines that refer to the named table
- You must have privileges to delete and define tables and related objects.

- You can roll back the TRUNCATE TABLE statement.
- The TRUNCATE TABLE statement fails with an error message if:
  - RDB$SYSTEM storage area is set to read-only
  - The named table is a view
  - The named table has been reserved for data definfition
  - The named table is a system table
- The TRUNCATE TABLE statement is not supported on created or declared local temporary tables.
- When truncating a table that contains a column of data type LIST OF BYTE VARYING, each row of the table must be read so the LIST data can be deleted.
- All constraints that reference the truncated table are revalidated (as not deferrable) after the truncate operation to ensure that the database remains consistent.

  If constraint validation fails, the TRUNCATE statement is automatically rolled back. For example:

  ```
  SQL> CREATE TABLE test1
  cont> (col1 REAL);
  SQL>
  SQL> CREATE TABLE test2
  cont> (col1 REAL,
  cont> CHECK (col1 IN
  cont>   (SELECT col1 FROM test1))
  cont> );
  SQL> COMMIT;
  SQL>
  SQL> INSERT INTO test1 VALUES (1);
  1 row inserted
  SQL> INSERT INTO test2 VALUES (1);
  1 row inserted
  SQL> COMMIT;
  SQL> TRUNCATE TABLE test1;
  -RDB-E-INTEG_FAIL, violation of constraint TEST2_CHECK1 caused operation to
  fail
  -RDB-F-ON_DB, on database SYS$LOGIN:[TEST]MF_PERSONNEL.RDB;
  ```

- Truncating a table does not delete the workload information collected in the RDB$WORKLOAD system table. You can delete the obsolete data with the RMU Delete Optimizer_Statistics command. See the *Oracle RMU Reference Manual* for further details.

## Examples

Example 1: Deleting data from a table while still maintaining the metadata definitions

The following example shows how to delete the data from the SALARY_HISTORY table and still maintain the metadata definitions:

```
SQL> TRUNCATE TABLE salary_history;
SQL> --
SQL> -- The table still exists, but the rows are deleted.
SQL> --
SQL> SELECT * FROM salary_history;
0 rows selected
SQL> SHOW TABLE (COLUMN) salary_history;
Information for table SALARY_HISTORY

Columns for table SALARY_HISTORY:
Column Name                     Data Type        Domain
-----------                     ---------        ------
EMPLOYEE_ID                     CHAR(5)          ID_DOM
 Foreign Key constraint SALARY_HISTORY_FOREIGN1
SALARY_AMOUNT                   INTEGER(2)       SALARY_DOM
SALARY_START                    DATE VMS         DATE_DOM
SALARY_END                      DATE VMS         DATE_DOM
```

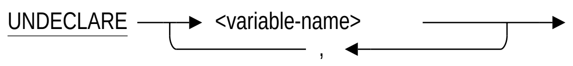
## UNDECLARE Variable Statement

Deletes a variable definition that you can use in interactive SQL for invoking stored procedures and for testing procedures in modules or embedded SQL programs.

### Environment

You can issue the UNDECLARE variable statement only in interactive SQL.

### Format



### Arguments

**variable-name**
Specifies the name of the local variables for interactive SQL.

### Example

Example 1: Undeclaring variables in interactive SQL

```
SQL> ATTACH 'FILENAME personnel';
SQL>
SQL> DECLARE :X INTEGER;
SQL> DECLARE :Y CHAR(10);
SQL>
SQL> BEGIN
cont>   SET :X = 100;
cont>   SET :Y = 'Active';
cont> END;
SQL> PRINT :X, :Y;
           X   Y
         100   Active
SQL> SHOW VARIABLES
X                               INTEGER
Y                               CHAR(10)
SQL> UNDECLARE :X, :Y;
```

## UPDATE Statement

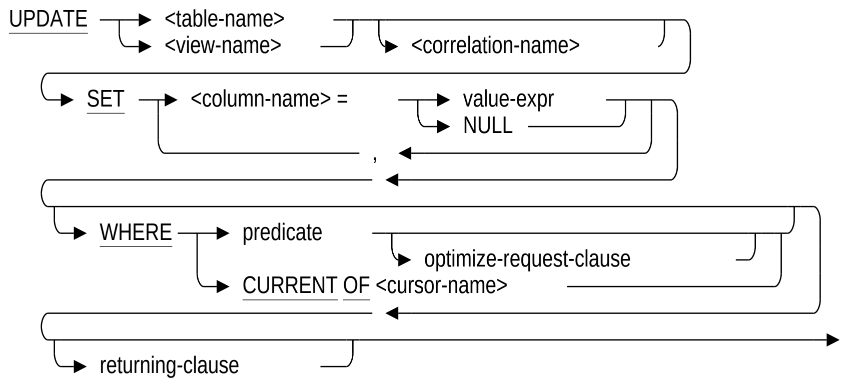
Modifies a row in a table or view.

### Environment

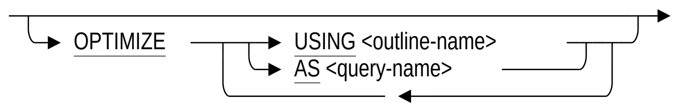
You can use the UPDATE statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- As part of a procedure in an SQL module
- In dynamic SQL as a statement to be dynamically executed

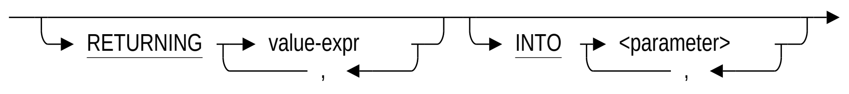
### Format



optimize-request-clause =



returning-clause=



## Arguments

**table-name**
**view-name**
Specifies the name of the target table or view that you want to modify.

**correlation-name**
Specifies a name you can use to identify the table or view in the predicate of the UPDATE statement. See Section 2.2.10.1 for more information about correlation names.

**SET**
Specifies which columns in the table or view get what values. For each column you want to modify, you must specify the column name and either a value expression or the NULL keyword. SQL assigns the value following the equal sign to the column that precedes the equal sign.

**column-name**
Specifies the name of a column whose value you want to modify.

**value-expr**
Specifies the new value for the modified column. Columns named in the value expression must be columns of the table or view named after the UPDATE keyword. The value expression can include functions like SUBSTRING, CAST, and POSITION (and so on) and external functions. The value expression cannot include functions like MAX, MIN, AVG, COUNT, and SUM.

**NULL**
Specifies a NULL keyword. SQL assigns a null value to columns for which you specify NULL. Any column assigned a null value must be defined to allow null values (defined in a CREATE or ALTER TABLE statement without the NOT NULL clause).

**WHERE**
Specifies the rows of the target table or view that will be modified according to the values indicated in the SET clause. If you omit the WHERE clause, SQL modifies all rows of the target table or view. You can specify either a predicate or a cursor name in the WHERE clause.

**predicate**
If the WHERE clause includes a predicate, all the rows of the target table for which the predicate is true are modified.

The columns named in the predicate must be columns of the target table or view. The target table cannot be named in a column select expression within the predicate.

See Section 2.7 for more information on predicates.

**OPTIMIZE USING outline-name**
Explicitly names the query outline to be used with the UPDATE statement even if the outline ID for the query and for the outline are different.

**OPTIMIZE AS query-name**
Assigns a name to the query.

**CURRENT OF cursor-name**
If the WHERE clause uses CURRENT OF cursor-name, SQL modifies only the row on which the named cursor is positioned. The cursor named in an UPDATE statement must meet these conditions:

- The cursor must have been named previously in a DECLARE CURSOR statement or FOR statement.
- The cursor must be open.
- The cursor must be on a row.
- The FROM clause of the SELECT statement within the DECLARE CURSOR statement must refer to the table or view that is the target of the UPDATE statement.

**RETURNING value-expr**
Returns the value of the column specified in the value expression. If DBKEY is specified, SQL returns the database key (dbkey) of the row being updated. When the DBKEY value is valid, subsequent queries can use the DBKEY value to access the row directly.

The RETURNING DBKEY clause is not valid in an UPDATE statement used to assign values to the segments in a column of the LIST OF BYTE VARYING data type.

Only one row can be updated when you specify the RETURNING clause.

**INTO parameter**
Inserts the value specified to a specified parameter. The INTO parameter clause is optional in interactive SQL.

## Usage Notes

- When you use the UPDATE statement to modify rows in a view, you change the rows of the base tables on which the view is based. Because of this, you cannot use the UPDATE statement on all views. See the CREATE VIEW Statement for rules about inserting, updating, and deleting values in views.
- SQL does not require UPDATE statements that specify WHERE CURRENT OF to refer to cursors declared with the appropriate FOR UPDATE clause.
  - If you specify columns in the SET clause that are not in the FOR UPDATE clause, SQL issues a warning message and proceeds with the update modifications.
  - If there is no FOR UPDATE clause with the DECLARE CURSOR statement, you can update any column. SQL will not issue any messages.
- The CURRENT OF clause in an embedded UPDATE statement cannot name a cursor based on a dynamic SELECT statement. To refer to a cursor based on a dynamic SELECT statement in the CURRENT OF clause, you must prepare and dynamically execute the UPDATE statement as well.
- The CURRENT OF clause in an embedded UPDATE statement cannot name a read-only cursor. See the DECLARE CURSOR Statement for Usage Notes about read-only cursors.
- When specifying a column name in the UPDATE statement, if the column name is the same as a parameter, you must use a correlation name or table name with the column name.
- You cannot specify the OPTIMIZE USING or the OPTIMIZE AS clause with the WHERE CURRENT OF clause.
- You cannot specify an outline name in a compound-use-statement. See the Compound Statement for more information about compound statements.
- If an outline exists, Oracle Rdb will use the outline specified in the OPTIMIZE USING clause unless one or more of the directives in the outline cannot be followed. SQL issues an error message if the existing outline cannot be used.

If you specify the name of an outline that does not exist, Oracle Rdb compiles the query, ignores the outline name, and searches for an existing outline with the same outline ID as the query. If an outline with the same outline ID is found, Oracle Rdb attempts to execute the query using the directives in that outline. If an outline with the same outline ID is not found, the optimizer selects a strategy for the query for execution.

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information regarding query outlines.

## Examples

Example 1: Using the UPDATE statement in interactive SQL

The following interactive SQL example changes the address of the employee with EMPLOYEE_ID 00164 and confirms the change:

```
SQL> UPDATE EMPLOYEES
cont>   SET ADDRESS_DATA_1 = '16 Ridge St.'
cont>   WHERE EMPLOYEE_ID = '00164';
1 row updated
SQL> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, ADDRESS_DATA_1
cont>   FROM EMPLOYEES
cont>   WHERE EMPLOYEE_ID = '00164';
 EMPLOYEE_ID   FIRST_NAME   LAST_NAME       ADDRESS_DATA_1
 00164         Alvin        Toliver         16 Ridge St.
1 row selected
```

Example 2: Using the UPDATE statement in a program

The following example illustrates using a host language variable in an embedded SQL statement to update an employee's status code:

```
   DISPLAY "Enter employee's ID number: " WITH NO ADVANCING.
   ACCEPT ID.
   DISPLAY "Enter new status code: " WITH NO ADVANCING.
   ACCEPT STATUS-CODE.

EXEC SQL
        DECLARE TRANSACTION READ WRITE
END-EXEC

EXEC SQL
        UPDATE EMPLOYEES
                SET STATUS_CODE = :STATUS-CODE
                WHERE EMPLOYEE_ID = :ID
END-EXEC

EXEC SQL  COMMIT  END-EXEC
```

## WHENEVER Statement

Specifies the execution path a host language program will take when any embedded SQL statement results in one of these following exception conditions:

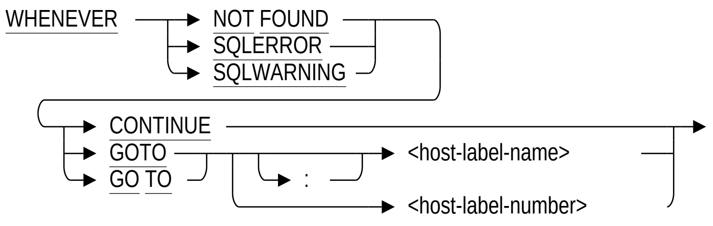
- Row not found
- An error condition
- A warning condition

For these conditions, the WHENEVER statement specifies that the program continue execution or branch to another part of the program.

### Environment

You can issue the WHENEVER statement only in host language programs.

### Format



### Arguments

**NOT FOUND**
Indicates the exception condition returned when SQL processes all the rows of a result table:

- When a cursor referred to in a FETCH, UPDATE, or DELETE statement is positioned after the last row
- When a query specifies an empty result table

This is the same condition identified by a value of 100 in the SQLCODE variable, the value of '02000' in the SQLSTATE variable, and by the RDB$_ STREAM_EOF error.

**SQLERROR**
Indicates any error condition. For the SQLERROR argument of the WHENEVER statement, SQL defines an error condition as any condition that returns a negative value to SQLCODE. See Appendix B for a list of the conditions that result in negative values for the SQLCODE field.

**SQLWARNING**
Indicates any warning condition. See Appendix B for a list of the conditions that result in warnings for the SQLCODE field. Appendix C lists the conditions that result in warnings for the SQLSTATE Status Parameter.

**CONTINUE**
Specifies that the program continue execution with the next sequential statement following the statement that generated an error.

**GOTO host-label-name**
**GOTO host-label-number**
Specifies that the program branch to the statement identified by the host label. The form of the host label depends on the host language. You can use a colon (:) before a host label represented by a name, but not before a host label represented by a number.

## Usage Notes

- Use of WHENEVER statements is optional. Omitting a WHENEVER statement for a class of exception conditions is equivalent to specifying the CONTINUE argument for that class of conditions.
- WHENEVER statements are not executable. SQL evaluates WHENEVER statements when the program precompiles. This means that the scope of a given WHENEVER statement cannot be controlled by conditional statements in the host program. A given WHENEVER statement affects all executable SQL statements until the precompiler encounters the next WHENEVER statement for the same exception condition in its sequential processing of the source program.
- Once you specify a WHENEVER . . . GOTO statement for a class of exception conditions, you can disable it with a WHENEVER . . . CONTINUE statement for that class of conditions.
- The ANSI/ISO 1989 standard requires a colon (:) before the host label name in the GOTO clause. The ANSI/ISO SQL standard does not allow this colon.

## Example

OpenVMS VAX OpenVMS Alpha

Example 1: Using WHENEVER statements in a PL/I program

```
/* When an SQL statement results in
an RDB$_STREAM_EOF error, the
program branches to LABEL_NOT_FOUND: */
EXEC SQL WHENEVER NOT FOUND GOTO LABEL_NOT_FOUND;

/* When an SQL statement results in a
warning severity error condition, the
program branches to LABEL_ERROR: */
EXEC SQL WHENEVER SQLWARNING GOTO LABEL_ERROR;

/* When an SQL statement results in
an error severity exception condition, the
program branches to LABEL_ERROR: */
EXEC SQL WHENEVER SQLERROR GOTO LABEL_ERROR;
```

♦

# A
# Error Messages

This appendix describes:

- The types and format of error messages you can encounter when using SQL
- How to find and use the documentation for error messages

## A.1 Types of Error Messages and Their Format

You can receive messages not only from SQL, but also from underlying software.

Messages you encounter while using SQL come from the following levels:

- The SQL interface itself. Messages generated by SQL are preceded by a facility code of SQL. For example:

  ```
  %SQL-E-CURALROPE, Cursor K was already open
  ```

  In programs, you can use the message vector structure in the sql_signal, sql_get_error_text, sql_get_message_vector, and SQL$GET_ERROR_TEXT routines, described in Section B.2, to signal errors and return the corresponding message text.

- Common Operating System Interface (COSI) facility error messages. For example:

  ```
  %COSI-F-NOQUAL, qualifiers not allowed - supply only verb and parameters
  ```

- The underlying database product. The facility code for messages generated by the underlying database depends on the database product with which you are using SQL.

  Oracle Rdb messages are preceded by a facility code of RDMS. For example:

  ```
  %RDMS-F-INVDB_NAME, invalid database name
  ```

  Refer to the appropriate documentation for other products.

- The repository. Messages generated by the repository are preceded by a facility code of CDD. For example:

  ```
  %CDD-E-ERRSHOW, error displaying object
  ```

Whatever the source of an error message, the format is the same. All error messages contain the following elements:

- The facility name preceded by a percent sign (%) or a hyphen (-)
- The severity code followed by a hyphen (-)

  Table A–1 lists the severity codes in order of increasing severity.
- The diagnostic error message name followed by a comma (,)

  This name identifies the message. In the documentation for error messages, the messages are alphabetized within each facility by diagnostic error message name.
- The diagnostic error message text

  The text is a brief description of the problem. Error messages may contain string substitutions that are specific to a user's error. In the documentation for error messages, these string substitutions are delimited by angle brackets (< >) within a message. For example:

  ```
  %SQL-F-CURNOTOPE, Cursor <str> is not opened
  ```

  If you receive this message, SQL substitutes the actual string (in this case, a cursor name) for <str>.

OpenVMS VAX OpenVMS Alpha

You can suppress the display of any or all elements of an error message with the SET MESSAGE command in DCL. ♦

**Table A–1 Explanation of Error Message Severity Codes**

| Code | Severity | Explanation |
|---|---|---|
| S | Success | Indicates that your command executed successfully. |
| I | Information | Reports on actions taken by the software. |

(continued on next page)

**Table A–1 (Cont.) Explanation of Error Message Severity Codes**

| Code | Severity | Explanation |
|---|---|---|
| W | Warning | Indicates error conditions for which you may not need to take corrective action. |
| E | Error | Indicates conditions that are not fatal, but that must be handled or corrected. |
| F | Fatal | Indicates conditions that are fatal and must be handled or corrected. |

## A.2 Error Message Documentation

Because error messages are updated frequently, documentation is provided in the following online files:

- SQL messages:

  For OpenVMS, in SYS$HELP:SQL$MSG.DOC

  For Digital UNIX, in /usr/lib/dbs/sql/vnn/help/sql_msg.doc

  This file contains the same text as the Help Errors help topic in interactive SQL.

- RDB messages:

  For OpenVMS, in SYS$HELP:RDB_MSG.DOC

  For Digital UNIX, in /usr/lib/dbs/rdb/vnn/help/rdb_msg.doc

- RDMS messages:

  For OpenVMS, in SYS$HELP:RDMS_MSG.DOC

  For Digital UNIX, /usr/lib/dbs/rdb/vnn/help/rdms_msg.doc

- COSI messages:

  For OpenVMS, in SYS$HELP:COSI_MSG.DOC

  For Digital UNIX, in /usr/lib/dbs/sql/vnn/help/cosi_msg.doc

- SQL/Services messages:

  For OpenVMS, in SYS$HELP:SQLSRV$MSG.DOC

  For Digital UNIX, in /usr/lib/dbs/sqs/vnn/doc/sqs_messages.txt and /usr/lib/dbs/sqs/vnn/doc/sqs_messages.ps

OpenVMS VAX OpenVMS Alpha

- Repository messages:

  SYS$HELP:CDDPLUS_MSG.DOC ♦

The message documentation for all the facilities follows the same format, with messages alphabetized by message name. After the message name and text, the documentation includes an explanation and suggested user action.

The online message documentation files may be updated even if you do not install a new version of SQL. In particular, any installation of Oracle Rdb databases may replace the RDB_MSG.DOC file with one that is more up-to-date.

You can print the online message documentation files for reference. In addition, you can search the files for the message information you need.

## A.3 Errors Generated When You Use SQL Statements

When you write application programs that use SQL, you must use one of the following methods to return the error messages:

- The SQLCODE parameter, which stores a value that represents the execution status of SQL statements.
- The longword array RDB$MESSAGE_VECTOR, which stores information about the execution status of SQL statements.
- The calls sql_signal, sql_get_error_text, and SQL$GET_ERROR_TEXT, which use error information returned through the RDB$MESSAGE_VECTOR array.
- The call sql_get_message_vector, which retrieves information from the message vector about the status of the last SQL statement.
- The SQL statement WHENEVER, which provides error handling for all SQL statements that follow the WHENEVER statement. (However, you cannot use this statement in programs that call procedures in an SQL module.)
- The SQLSTATE status parameter, a string of five characters, provides error handling that complies with the ANSI/ISO SQL standard. See Appendix C for more information on the SQLSTATE status parameter.

For more information about handling errors using SQL options, see the *Oracle Rdb7 Guide to SQL Programming*.

Table A–2 lists SQL statements and errors they commonly generate at run time. This is not an exhaustive list. The second column lists the error message status code and the third column lists the corresponding value of the SQLCODE field in the SQLCA. See Appendix B for more information about SQLCODE values.

**Table A–2 SQL Errors Generated at Run Time**

| SQL Statement | Error Status Code[2] | SQLCODE Value |
|---|---|---|
| ALTER DOMAIN | SQL$_BAD_LENGTH | –1029 |
| | SQL$_BAD_SCALE | –1030 |
| | SQL$_NO_SUCH_FIELD | –1027 |
| ALTER TABLE | RDB$_DEADLOCK | –913 |
| | RDB$_INTEG_FAIL | –1001 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | RDB$_NO_PRIV | –1028 |
| | RDB$_READ_ONLY_REL | –1031 |
| | RDB$_READ_ONLY_TRANS | –817 |
| | RDB$_READ_ONLY_VIEW | –1031 |
| | RDB$_REQ_NO_TRANS | Not available[1] |
| | SQL$_BAD_LENGTH | –1029 |
| | SQL$_BAD_SCALE | –1030 |
| | SQL$_COLEXISTS | –1023 |
| | SQL$_FLDNOTDEF | –1024 |
| | SQL$_FLDNOTINREL | –1024 |
| | SQL$_NO_SUCH_FIELD | –1027 |
| ATTACH | RDB$_REQ_WRONG_DB | –1020 |
| CLOSE | SQL$_CURNOTOPE | –501 |
| COMMIT | RDB$_DEADLOCK | –913 |
| | RDB$_INTEG_FAIL | –1001 |
| | RDB$_LOCK_CONFLICT | –1003 |

[1]No specific numeric value. Check the SQLCODE for negative values.

[2]-1 is a general error SQLCODE value that does not correspond to any specific error. Use sql_ signal or sql_get_error_text to return a meaningful error.

(continued on next page)

**Table A–2 (Cont.) SQL Errors Generated at Run Time**

| SQL Statement | Error Status Code[2] | SQLCODE Value |
|---|---|---|
| | SQL$_NO_TXNOUT | –1005 |
| CREATE DOMAIN | SQL$_FIELD_EXISTS | –1026 |
| CREATE VIEW | RDB$_DEADLOCK | –913 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | SQL$_NO_SUCH_FIELD | –1027 |
| | SQL$_REL_EXISTS | –1025 |
| DELETE | RDB$_DEADLOCK | –913 |
| | RDB$_INTEG_FAIL | –1001 |
| | RDB$_LOCK_CONFLICT | –1003 |
| DELETE WHERE | RDB$_DEADLOCK | –913 |
| CURRENT | RDB$_INTEG_FAIL | –1001 |
| | SQL$_CURNOTOPE | –501 |
| | SQL$_FETNOTDON | –508 |
| FETCH | RDB$_DEADLOCK | –913 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | RDB$_STREAM_EOF | 100 |
| | SQL$_CURNOTOPE | –501 |
| | SQL$_NULLNOIND | –305 |
| INSERT | RDB$_ARITH_EXCEPT | –304 |
| | RDB$_DEADLOCK | –913 |
| | RDB$_INTEG_FAIL | –1001 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | RDB$_NO_DUP | –803 |
| | RDB$_NO_PRIV | –1028 |
| | RDB$_NOT_VALID | –1002 |
| | RDB$_OBSOLETE_METADATA | –1032 |
| | RDB$_READ_ONLY_REL | –1031 |
| | RDB$_READ_ONLY_TRANS | –817 |

[2]-1 is a general error SQLCODE value that does not correspond to any specific error. Use sql_ signal or sql_get_error_text to return a meaningful error.

(continued on next page)

**Table A–2 (Cont.) SQL Errors Generated at Run Time**

| SQL Statement | Error Status Code[2] | SQLCODE Value |
|---|---|---|
| | RDB$_READ_ONLY_VIEW | –1031 |
| | RDB$_REQ_NO_TRANS | Not available[1] |
| | RDB$_REQ_WRONG_DB | –1020 |
| | RDB$_UNRES_REL | –1033 |
| OPEN | RDB$_DEADLOCK | –913 |
| | RDB$_LOCK_CONFLICT | –1003 |
| ROLLBACK | SQL$_NO_TXNOUT | –1005 |
| SET TRANSACTION | RDB$_DEADLOCK | –913 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | SQL$_BAD_TXN_STATE | –1004 |
| singleton SELECT | RDB$_DEADLOCK | –913 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | SQL$_NULLNOIND | –305 |
| | SQL$_SELMORVAL | –811 |
| UPDATE | RDB$_DEADLOCK | –913 |
| | RDB$_INTEG_FAIL | –1001 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | RDB$_NO_DUP | –803 |
| | RDB$_NOT_VALID | –1002 |
| | RDB$_READ_ONLY_REL | –1031 |
| UPDATE WHERE | RDB$_DEADLOCK | –913 |
| CURRENT | RDB$_INTEG_FAIL | –1001 |
| | RDB$_LOCK_CONFLICT | –1003 |
| | RDB$_NO_DUP | –803 |
| | RDB$_NOT_VALID | –1002 |
| | SQL$_CURNOTOPE | –501 |
| | SQL$_FETNOTDON | –508 |

[1]No specific numeric value. Check the SQLCODE for negative values.

[2]-1 is a general error SQLCODE value that does not correspond to any specific error. Use sql_signal or sql_get_error_text to return a meaningful error.

## A.4 Identifying Precompiler and Module Language Errors

The SQL precompiler and the SQL module language processor let you identify (flag) syntax that is not ANSI/ISO SQL standard. See Chapter 3 and Chapter 4 for more information.

Error messages for SQL precompilers and SQL module language are flagged in the following way:

```
EXEC SQL SELECT SUM(DISTINCT QTY), AVG(DISTINCT QTY)    /* multiple distincts*/
%SQL-I-NONSTADIS, (1) The standard only permits one DISTINCT in a select expression
    INTO :int1, :int2 FROM D.SP;            /*    in a query     */
```

## A.5 sqlmsg.mdf - Error Message File for Digital UNIX

Digital UNIX

The sqlmsg.mdf message file is located in the default /usr/lib/dbs/sql/vnn/lib/C_C subdirectory.

**Locale** refers to the international environment of a computer program defining localized behavior of that program at run time. This information can be established from one or more sets of local databases.

If the name of the current locale MESSAGES category is C (that is, it has not previously been defined), then the value of the locale name is determined by the following precedence order. The first condition met determines the value:

1. LC_ALL environment variable is defined and is not null.
2. LC_MESSAGES environment variable is defined and is not null.
3. LANG environment variable is defined and is not null.
4. LANG environment variable is not set or is set to an empty string then the C locale is used.

For example, if LC_ALL = "ja_JP.deckanji" then Japanese messages will be used and the message file will be scanned for in the following order:

```
/usr/lib/dbs/sql/vnn/lib/ja_JP/sqlmsg.mdf
/usr/lib/dbs/sql/vnn/lib/ja_C/sqlmsg.mdf
/usr/lib/dbs/sql/vnn/lib/C_C/sqlmsg.mdf
```

♦

# B
# The SQL Communications Area (SQLCA) and the Message Vector

The SQLCA and message vector are two separate host structures that SQL declares when it precompiles an INCLUDE SQLCA statement.

Both the SQLCA and the message vector provide ways of handling errors:

- The SQLCA is a collection of parameters that SQL uses to provide information about the execution of SQL statements to application programs. The SQLCODE parameter in the SQLCA shows if a statement was successful and, for some errors, the particular error when a statement is not successful.

  To illustrate how the SQLCA works in applications, interactive SQL displays its contents when you issue the SHOW SQLCA statement.

- The message vector is also a collection of parameters that SQL updates after it executes a statement. It lets programs check if a statement was successful, but provides more detail than the SQLCA about the type of error if a statement is not successful. The message vector, for example, provides a way to access any follow-on messages in addition to those containing the facility code RDB or SQL.

  You can use the following steps to examine the message vector:

  - Assign any value to the logical name SQL$KEEP_PREP_FILES or configuration parameter SQL_KEEP_PREP_FILES.
  - Precompile any program that contains the line “EXEC SQL INCLUDE SQLCA”. (You can use the programs in the sample directory.)
  - Examine the generated host language program.

SQL updates the contents of the SQLCA and the message vector after completion of every executable SQL statement (nonexecutable statements are the DECLARE, WHENEVER, and INCLUDE statements).

You do not have to use the INCLUDE SQLCA statement in programs. However, if you do not, you must explicitly declare the SQLCODE parameter to receive values from SQL. SQLCODE is explicitly declared as an unscaled, signed longword integer.

SQLCODE is a deprecated feature of the ANSI/ISO SQL standard and is replaced by SQLSTATE. To comply with the ANSI/ISO SQL standard, you should explicitly declare either SQLCODE or, preferably, SQLSTATE instead of using the INCLUDE SQLCA statement. SQLCA (and the INCLUDE SQLCA statement) is not part of the ANSI/ISO SQL standard. If you declare SQLCODE or SQLSTATE but use the INCLUDE SQLCA statement, SQL uses the SQLCA.

Programs that do not use the INCLUDE SQLCA statement will not have the message vector declared by the precompiler. Such programs must explicitly declare the message vector if they:

- Use the RDB$LU_STATUS field of the message vector in their error checking
- Use system calls such as SYS$PUTMSG

The message vector is not part of the ANSI/ISO SQL standard.

When the SQLCA structure is explicitly declared by a program, SQL does not update the SQLERRD fields. If you want the SQLERRD fields updated, include the SQLCA definitions in the program using the EXEC SQL INCLUDE SQLCA statement.

Section B.1 and Section B.2 describe the SQLCA and the message vector in more detail. Section B.3 shows the declarations SQL makes for them in different host language programs.

## B.1 The SQLCA

The only fields of interest in the SQLCA are the SQLCODE field and the second through sixth elements of the SQLERRD array.

Example B–1 shows the interactive SQL display for the SQLCA after the “attempt to fetch past end of stream” error.

**Example B–1 Fields in the SQLCA**

```
SQL> SHOW SQLCA
SQLCA:
        SQLCAID:        SQLCA           SQLCABC:        128
        SQLCODE:        100
        SQLERRD:        [0]: 0
                        [1]: 0
                        [2]: 0
                        [3]: 0
                        [4]: 0
                        [5]: 0
        SQLWARN0:       0       SQLWARN1:       0       SQLWARN2:       0
        SQLWARN3:       0       SQLWARN4:       0       SQLWARN5:       0
        SQLWARN6:       0       SQLWARN7:       0
        SQLSTATE:       02000
```

SQLSTATE is not part of the SQLCA, although it appears in the display.

The remainder of this section describes the fields of the SQLCA.

### Fields of the SQLCA

#### SQLCAID

An 8-character field whose value is always the character string SQLCA. It is provided for compatibility with DB2 databases. The FORTRAN SQLCA does not include this field.

#### SQLCABC

An integer field whose value is always the length, in bytes, of the SQLCA. It is provided for compatibility with DB2 databases. The value is always 128. The FORTRAN SQLCA does not include this field.

#### SQLCODE

An integer field whose value indicates the error status returned by the most recently executed SQL statement. A positive value other than 100 indicates a warning, a negative value indicates an error, and a zero indicates successful execution.

Table B–1 shows the possible numeric and literal values that SQL returns to the SQLCODE field and explains the meaning of the values.

**Table B–1 Values Returned to the SQLCODE Field**

| Numeric Value | Literal Value | Meaning |
|---|---|---|
| **Success Status Code** | | |
| 0 | SQLCODE_SUCCESS | Statement completed successfully. |
| **Warning Status Codes** | | |
| 100 | SQLCODE_EOS | SELECT statement or cursor came to the end of stream. |
| 1003 | SQLCODE_ELIM_NULL[1] | Null value was eliminated in a set function. |
| 1004 | SQLCODE_TRUN_RTRV[1] | String truncated during assignment. This occurs only during data retrieval. |
| **Error Status Codes** | | |
| –1 | SQLCODE_RDBERR | Oracle Rdb returned an error. The value of –1 is a general error SQLCODE value returned by any error not corresponding to the other values in this table. Use sql_signal or sql_get_error_text to return a meaningful error. |
| –304 | SQLCODE_OUTOFRAN | Value is out of range for a host variable. |
| –305 | SQLCODE_NULLNOIND | Tried to store a null value into a host language variable with no indicator variable. |
| –306 | SQLCODE_STR_DAT_ TRUNC[1] | String data, right truncation. |
| –307 | SQLCODE_INV_DATETIME | Date-time format is invalid. |
| –501 | SQLCODE_CURNOTOPE | Cursor is not open. |
| –502 | SQLCODE_CURALROPE | Cursor is already open. |
| –507 | SQLCODE_UDCURNOPE | Cursor in an UPDATE or DELETE operation is not opened. |
| –508 | SQLCODE_UDCURNPOS | Cursor in an UPDATE or DELETE operation is not positioned on a row. |
| –509 | SQLCODE_UDCURDEL | Cursor in an UPDATE or DELETE operation is positioned on a deleted row. |

[1]Only the SQL92 dialect returns this value.

(continued on next page)

**Table B–1 (Cont.) Values Returned to the SQLCODE Field**

| Numeric Value | Literal Value | Meaning |
|---|---|---|
| **Error Status Codes** | | |
| –803 | SQLCODE_NO_DUP | Updating would cause duplication on a unique index. |
| –811 | SQLCODE_SELMORVAL | The result of a singleton select returned more than one value. |
| –817 | SQLCODE_ROTXN | Attempt to update from a read-only transaction. |
| –913 | SQLCODE_DEADLOCK | Request failed due to resource deadlock. |
| –1001 | SQLCODE_INTEG_FAIL | Constraint failed. |
| –1002 | SQLCODE_NOT_VALID | Valid-if failed. |
| –1003 | SQLCODE_LOCK_ CONFLICT | NO WAIT request failed because resource was locked. |
| –1004 | SQLCODE_BAD_TXN_ STATE | Invalid transaction state–the transaction already started. |
| –1005 | SQLCODE_NO_TXN | No transaction active. |
| –1006 | SQLCODE_BAD_VERSION | Version of the underlying system does not support a feature that this query uses. |
| –1007 | SQLCODE_TRIG_ERROR | Trigger forced an error. |
| –1008 | SQLCODE_NOIMPTXN | No implicit distributed transaction outstanding. |
| –1009 | SQLCODE_DISTIDERR | Distributed transaction ID error. |
| –1010 | SQLCODE_BAD_CTX_VER | Version field in the context structure is defined incorrectly. |
| –1011 | SQLCODE_BAD_CTX_ TYPE | Type field in the context structure is defined incorrectly. |
| –1012 | SQLCODE_BAD_CTX_LEN | Length field in the context structure is defined incorrectly. |
| –1013 | SQLCODE_BASROWDEL | Row that contains the list has been deleted. |
| –1014 | SQLCODE_DIFFDEFINV | Invoker of the module is not the same as the definer (the user who compiled the module). |
| –1015 | SQLCODE_STMTNOTPRE | Dynamic statement is not prepared. |
| –1016 | SQLCODE_NOSUCHCONN | Connection does not exist. |

(continued on next page)

**Table B–1 (Cont.) Values Returned to the SQLCODE Field**

| Numeric Value | Literal Value | Meaning |
|---|---|---|
| **Error Status Codes** | | |
| –1017 | SQLCODE_CONNAMEXI | Connection name already exists. |
| –1018 | SQLCODE_DBENVSYNERR | Database environment specification contains a syntax error. |
| –1019 | SQLCODE_DBSPECSYNERR | Database specification contains a syntax error. |
| –1020 | SQLCODE_ATTACHERR | Error attaching to the database. |
| –1021 | SQLCODE_NOSUCHALIAS | Alias is not known. |
| –1022 | SQLCODE_ALIASINUSE | Alias is already declared. |
| –1023 | SQLCODE_COLEXISTS | Column already exists in the table. |
| –1024 | SQLCODE_COLNOTDEF | Column not defined in the table. |
| –1025 | SQLCODE_TBLEXISTS | Table already exists in the database or schema. |
| –1026 | SQLCODE_DOMEXISTS | Domain already exists in the database or schema. |
| –1027 | SQLCODE_DOMNOTDEF | Domain is not defined in the database or schema. |
| –1028 | SQLCODE_NO_PRIV | No privilege for attempted operation. |
| –1029 | SQLCODE_BAD_LENGTH | Negative length specified for a column. |
| –1030 | SQLCODE_BAD_SCALE | Negative scale specified for a column. |
| –1031 | SQLCODE_RO_TABLE | Attempt to update a read-only table. |
| –1032 | SQLCODE_OBSMETADATA | Metadata no longer exists. |
| –1033 | SQLCODE_UNRES_REL | Table is not reserved in the transaction. |
| –1034 | SQLCODE_CASENOTFND | Case not found; WHEN not specified. |
| –1035 | SQLCODE_CHKOPT_VIOL | Integer failure with check option. |
| –1036 | SQLCODE_UNTERM_C_ STR | Unterminated C string. |
| –1037 | SQLCODE_INDIC_ OVFLOW | Indicator overflow. |
| –1038 | SQLCODE_INV_PARAM_ VAL | Invalid parameter value. |

(continued on next page)

**Table B–1 (Cont.) Values Returned to the SQLCODE Field**

| Numeric Value | Literal Value | Meaning |
|---|---|---|
| **Error Status Codes** | | |
| –1039 | SQLCODE_NULL_ELIMIN | Null eliminated in the set function. |
| –1040 | SQLCODE_INV_ESC_SEQ | Invalid escape sequence. |
| –1041 | SQLCODE_RELNOTDEF | Table not defined in the database or schema. |

Programs can use the literal values to check for success, the end of record stream warnings, or specific errors. Your program can check for particular error codes and execute different sets of error-handling statements depending upon the error code returned. However, because the values in Table B–1 do not reflect all the possible errors or warnings, your program should check for *any* negative value.

SQL inserts the RDB message vector (see Section B.2) along with the SQLCA structure when it executes an SQL statement.

Also, string truncation conditions are only reported when the dialect is set to SQL92 prior to a database attach in interactive SQL or when your application is compiled. For example:

```
SQL> SET DIALECT 'SQL92';
SQL> ATTACH 'FILENAME mf_personnel';
SQL> DECLARE :ln CHAR(10);
SQL> SELECT last_name INTO :ln FROM employees WHERE employee_id = '00164';
%RDB-I-TRUN_RTRV, string truncated during assignment to a variable or parameter
SQL> SHOW SQLCA
SQLCA:
        SQLCAID:        SQLCA           SQLCABC:        128
        SQLCODE:        1004
        SQLERRD:        [0]: 0
                        [1]: 0
                        [2]: 1
                        [3]: 0
                        [4]: 0
                        [5]: 0
        SQLWARN0:       0       SQLWARN1:       0       SQLWARN2:       0
        SQLWARN3:       0       SQLWARN4:       0       SQLWARN5:       0
        SQLWARN6:       0       SQLWARN7:       0
        SQLSTATE:       01004
%RDB-I-TRUN_RTRV, string truncated during assignment to a variable or parameter
```

For each language, SQL provides a file that contains the declarations of all the error literals shown in Table B–1. You can include this file in precompiled SQL and module language programs.

Table B–2 shows how to include this file in your program.

**Table B–2 Including the Error Literals File in Programs**

| Precompiled or Module Language | Declaration |
|---|---|
| Ada | with SQL_SQLCODE;<br>with SQL_SQLDA;<br>with SQL_SQLDA2; [1] |
| BASIC | %INCLUDE "sys$library:sql_literals.bas" |
| C | #include "sys$library:sql_literals.h" |
| COBOL | COPY 'SYS$LIBRARY:SQL_LITERALS' |
| FORTRAN | INCLUDE 'SYS$LIBRARY:SQL_LITERALS.FOR' |
| Pascal | %include 'sys$library:sql_literals.pas' |
| PL/I | %INCLUDE 'sys$library:sql_literals.pli'; |

[1]You must compile the Ada package, SYS$LIBRARY:SQL_LITERALS.ADA, before you use it in a program. Only declare SQL_SQLDA and SQL_SQLDA2 when you use dynamic SQL.

In addition to the error literals, the file contains declarations for the SQLTYPE field in the SQLDA. See Appendix D for information about the SQLTYPE field.

Example B–2 shows how to include the error literals file in a COBOL program.

**Example B–2 Including Error Literals in a COBOL Program**

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LITERAL-TESTS.
*
* This program tests the use of symbolic literals for SQLCODE and
* SQLDA_DATATYPE.  All the literal definitions are part of a file that
* is used with the COPY command.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY SQL_LITERALS.
EXEC SQL INCLUDE SQLCA END-EXEC.
01 CDE         PIC X(5).
01 DISP_SQLCODE PIC S9(9) DISPLAY SIGN LEADING SEPARATE.
01 GETERRVARS.
           02  error-buffer-len        PIC S9(9) COMP VALUE 132.
           02  error-msg-len           PIC S9(9) COMP.
           02  error-buffer            PIC X(132).

exec sql whenever sqlerror continue end-exec.

PROCEDURE DIVISION.

*
* test for sqlcode -501 SQLCODE_CURNOTOPE
*
        exec sql declare A cursor for
                select college_code from colleges
                where college_name like 'D%' order by 1
        end-exec.
        exec sql fetch A into :CDE end-exec.
        if sqlcode = SQLCODE_CURNOTOPE
        then
            MOVE sqlcode to DISP_SQLCODE
            DISPLAY "SQLCODE after attempt to fetch is ", DISP_SQLCODE
        CALL "sql_get_error_text" USING BY REFERENCE error-buffer,
                                        BY VALUE error-buffer-len,
                                        BY REFERENCE error-msg-len.
            DISPLAY BUFFER(1:error-msg-len)
        end-if.
        exec sql close A end-exec.
*
```

(continued on next page)

**Example B–2 (Cont.) Including Error Literals in a COBOL Program**

```
* test for SQLCODE 0 SQLCODE_SUCCESS
*
        exec sql
        insert into employees (employee_id, last_name, sex)
            values ('00999','Jones','M')
        end-exec.
        if sqlcode = SQLCODE_SUCCESS
        then
            MOVE sqlcode to DISP_SQLCODE
            DISPLAY "SQLCODE after insert is ", DISP_SQLCODE
        CALL "sql_get_error_text" USING BY REFERENCE error-buffer,
                                        BY VALUE error-buffer-len,
                                        BY REFERENCE error-msg-len.
            DISPLAY BUFFER(1:error-msg-len)
        end-if.

        EXEC SQL ROLLBACK END-EXEC.
        STOP RUN.
```

### SQLERRM

The SQLERRM is a structure containing two fields: a word field called SQLERRML and a 70-character field called SQLERRMC. It is provided only for compatibility with DB2 software.

### SQLERRD[x]

A zero-based array of six integers. The only elements of the array that SQL uses are the second through sixth elements (SQLERRD[1], SQLERRD[2], SQLERRD[3], SQLERRD[4] and SQLERRD[5] in the display from SHOW SQLCA). The remainder of the elements are provided for compatibility with DB2 software.

When you use dynamic SQL, SQL puts a value in the second element (SQLERRD[1]) after SQL executes the DESCRIBE statement. The values represent the following:

- 0: The statement is any SQL statement except a SELECT statement or CALL statement.
- 1: The statement is a SELECT statement.
- 2: The statement is a CALL statement.

SQL puts a value in the third element (SQLERRD[2]) after successful execution of the following statements:

- INSERT: The number of rows stored by the statement.

- UPDATE: The number of rows modified by the statement.
- DELETE: The number of rows deleted by the statement.
- FETCH: The number of the row on which the cursor is currently positioned.
- SELECT: The number of rows in the result table formed by the SELECT statement. (Note: The SQLERRD[2] field is not updated for dynamic SELECT statements.)

SQL puts the following values in the third and fourth elements after successful execution of an OPEN statement for a table cursor:

- SQLERRD[2]: Estimated result table cardinality
- SQLERRD[3]: Estimated I/O operations

You must recompile application modules so that the new values in SQLERRD[2] and SQLERRD[3] can be returned.

SQL puts the following values in the second, fourth, fifth, and sixth elements after successful execution of an OPEN statement that opens a list cursor:

- SQLERRD[1]: Longword length of the longest actual segment
- SQLERRD[3]: Longword number of segments
- SQLERRD[4,5]: Two contiguous longwords contained a quadword number of total bytes

SQL puts no meaningful data in the sixth element of the SQLERRD array after successful execution of a FETCH statement.

SQLERRD[1] on a LIST cursor fetch returns the segment size in octets.

After error statements or any other cases, the value of SQLERRD is undefined.

**SQLWARNx**
A series of 1-character fields, numbered from 0 through 7, that SQL does not use. Provided for compatibility with DB2 software.

## B.2 The Message Vector

When SQL precompiles a program, it declares a host structure for the message vector immediately following the SQLCA. It calls the structure RDB$MESSAGE_VECTOR.

Programs most often use the message vector in two ways:

- By checking the message vector field RDB$LU_STATUS for the return status value from the last SQL statement. The program can either check the low-order bit of that field (successful if set) or use the entire field to determine the specific return status value.
- By using the message vector in the sql_signal and sql_get_error_text routines:
  - The sql_signal routine uses the message vector to signal the error to the OpenVMS condition handler.
  - The sql_get_error_text routine puts the message text corresponding to the return status value in the message vector into a buffer the program specifies.

  For more information about sql_signal and sql_get_error_text, see Chapter 5.

Figure B–1 summarizes the fields of the message vector.

**Figure B–1 Fields of the Message Vector**

RDB$MESSAGE_VECTOR

| | |
|---|---|
| RDB$LU_NUM_ARGUMENTS | Number of arguments in the vector |
| RDB$LU_STATUS | Number corresponding to return status for the condition |
| RDB$ALU_ARGUMENTS | An array containing information about FAO arguments and follow-on messages related to the primary message, if any |
| RDB$LU_ARGUMENTS [1] | Number of FAO arguments to primary message |
| . | Pointer to FAO arguments, if any |
| . | Return status for follow-on message, if any |
| . | Number of FAO arguments, for follow-on message, if any |

## B.3 Declarations of the SQLCA and the Message Vector

This section shows the SQLCA and message vector declarations for the host languages supported by the SQL precompiler and module processor.

OpenVMS VAX OpenVMS Alpha

Example B–3 shows the Ada SQLCA and message vector declaration.

**Example B–3 Ada SQLCA and Message Vector Declaration**

```
Package SQL_ADA_CURSOR is
TYPE SQL_TYPE_1 IS NEW STRING(1..6);
    type SQLERRM_REC is
        record
            SQLERRML : short_integer;
            SQLERRMC : string (1..70);
        end record;

    type SQLERRD_ARRAY is array (1..6) of integer;

   type SQLCA is
        record
            SQLCAID : string (1..8) := "SQLCA   ";
            SQLABC : integer := 128;
            SQLCODE : integer;
            SQLERRM : sqlerrm_rec;
            SQLERRD : sqlerrd_array;
            SQLWARN0 : character := ' ';
            SQLWARN1 : character := ' ';
            SQLWARN2 : character := ' ';
            SQLWARN3 : character := ' ';
            SQLWARN4 : character := ' ';
            SQLWARN5 : character := ' ';
            SQLWARN6 : character := ' ';
            SQLWARN7 : character := ' ';
            SQLEXT : string (1..8) := "        ";
        end record;

RDB_MESSAGE_VECTOR : SYSTEM.UNSIGNED_LONGWORD_ARRAY(1..20);
pragma PSECT_OBJECT(RDB_MESSAGE_VECTOR,"RDB$MESSAGE_VECTOR");
♦
```

OpenVMS VAX OpenVMS Alpha

Example B–4 shows the BASIC SQLCA and message vector declaration.

**Example B–4 BASIC SQLCA and Message Vector Declaration**

```
RECORD SQLCA_REC
    string SQLCAID = 8
    long SQLCABC
    long SQLCODE
    GROUP SQLERRM
      word SQLERRML
      string SQLERRMC = 70
    END GROUP SQLERRM
    long SQLERRD(5)
    string SQLWARN0 = 1
    string SQLWARN1 = 1
    string SQLWARN2 = 1
    string SQLWARN3 = 1
    string SQLWARN4 = 1
    string SQLWARN5 = 1
    string SQLWARN6 = 1
    string SQLWARN7 = 1
    string SQLEXT = 8
END RECORD SQLCA_REC

DECLARE SQLCA_REC SQLCA


RECORD  RDB$MESSAGE_VECTOR_REC
    long RDB$LU_NUM_ARGUMENTS
    long RDB$LU_STATUS
    GROUP RDB$ALU_ARGUMENTS(17) ! Arrays in BASIC are always relative
          long RDB$LU_ARGUMENT  ! to 0.  There are 18 array elements.
    END GROUP RDB$ALU_ARGUMENTS
END RECORD RDB$MESSAGE_VECTOR_REC

COMMON (RDB$MESSAGE_VECTOR) &
    RDB$MESSAGE_VECTOR_REC RDB$MESSAGE_VECTOR
```

♦

Example B–5 shows the C SQLCA and message vector declaration.

**Example B–5 C SQLCA and Message Vector Declaration**

```
struct
    {
        char SQLCAID[8];
        int SQLCABC;
        int SQLCODE;
        struct {
            short SQLERRML;
            char SQLERRMC[70];
               } SQLERRM;
        int SQLERRD[6];
        struct {
            char SQLWARN0[1];
            char SQLWARN1[1];
            char SQLWARN2[1];
            char SQLWARN3[1];
            char SQLWARN4[1];
            char SQLWARN5[1];
            char SQLWARN6[1];
            char SQLWARN7[1];
               } SQLWARN;
        char SQLEXT[8];
    } SQLCA = {         "SQLCA   ",
                        128, 0,
                        {0, ""},
                        {0,0,0,0,0,0},
                        {"", "", "", "", "", "", "", ""},
                        "" };
extern
struct Rdb$MESSAGE_VECTOR_str
RDB$MESSAGE_VECTOR;
```

Example B–6 shows the COBOL SQLCA and message vector declaration.

**Example B–6 COBOL SQLCA and Message Vector Declaration**

```
01      SQLCA   GLOBAL.
        02      SQLCAID PIC X(8) VALUE IS "SQLCA   ".
        02      SQLCABC PIC S9(9) COMP VALUE IS 128.
        02      SQLCODE PIC S9(9) COMP.
        02      SQLERRM.
                03      SQLERRML PIC S9(4) COMP VALUE IS 0.
                03      SQLERRMC PIC X(70).
        02      SQLERRD PIC S9(9) COMP  OCCURS 6 TIMES.
        02      SQLWARN.
                03      SQLWARN0 PIC X.
                03      SQLWARN1 PIC X.
                03      SQLWARN2 PIC X.
                03      SQLWARN3 PIC X.
                03      SQLWARN4 PIC X.
                03      SQLWARN5 PIC X.
                03      SQLWARN6 PIC X.
                03      SQLWARN7 PIC X.
        02      SQLEXT PIC X(8).

01 Rdb$MESSAGE_VECTOR EXTERNAL GLOBAL.
    03 Rdb$LU_NUM_ARGUMENTS     PIC S9(9) COMP.
    03 Rdb$LU_STATUS            PIC S9(9) COMP.
    03 Rdb$ALU_ARGUMENTS        OCCURS 18 TIMES.
        05 Rdb$LU_ARGUMENTS     PIC S9(9) COMP.
```

Example B–7 shows the FORTRAN SQLCA and message vector declaration.

**Example B–7 FORTRAN SQLCA and Message Vector Declaration**

```
        CHARACTER*1 SQLCA (128)
        INTEGER*4 SQLCOD
        EQUIVALENCE (SQLCOD, SQLCA(13))
        INTEGER*2 SQLTXL
        EQUIVALENCE (SQLTXL, SQLCA(17))
        CHARACTER*70 SQLTXT
        EQUIVALENCE (SQLTXT, SQLCA(19))
        INTEGER*4 SQLERR(1:6)
        EQUIVALENCE (SQLERR, SQLCA(89))
        CHARACTER*1 SQLWRN(0:7)
        EQUIVALENCE (SQLWRN, SQLCA(113))
```

(continued on next page)

**Example B–7 (Cont.) FORTRAN SQLCA and Message Vector Declaration**

```
  INTEGER*4 Rdb$MESSAGE_VECTOR(20), Rdb$LU_NUM_ARGUMENTS
  INTEGER*4 Rdb$LU_STATUS, Rdb$ALU_ARGUMENTS(18)
  COMMON /Rdb$MESSAGE_VECTOR/ Rdb$MESSAGE_VECTOR
  EQUIVALENCE ( Rdb$MESSAGE_VECTOR(1),Rdb$LU_NUM_ARGUMENTS)
  EQUIVALENCE ( Rdb$MESSAGE_VECTOR(2), Rdb$LU_STATUS)
  EQUIVALENCE ( Rdb$MESSAGE_VECTOR(3), Rdb$ALU_ARGUMENTS )
```

Example B–8 shows the Pascal SQLCA and message vector declaration.

**Example B–8 Pascal SQLCA and Message Vector Declaration**

```
TYPE
 RDB$LU_ARGUMENTS = [HIDDEN] INTEGER;
 RDB$ALU_ARGUMENTS_ARRAY = [HIDDEN] ARRAY [1..18] OF RDB$LU_ARGUMENTS;
 RDB$MESSAGE_VECTOR_REC = [HIDDEN] RECORD
        RDB$LU_NUM_ARGUMENTS    :   INTEGER;
        RDB$LU_STATUS           :   INTEGER;
        RDB$ALU_ARGUMENTS       :   RDB$ALU_ARGUMENTS_ARRAY;
 END;
VAR
 RDB$MESSAGE_VECTOR : [HIDDEN, common(rdb$message_vector) ]
 RDB$MESSAGE_VECTOR_REC;
TYPE
 SQL$SQLCA_REC = [HIDDEN] RECORD
        SQLCAID : PACKED ARRAY [1..8] OF CHAR;
        SQLCABC : INTEGER;
        SQLCODE : INTEGER;
        SQLERRM : RECORD
            SQLERRML : SQL$SMALLINT;
            SQLERRMC : PACKED ARRAY [1..70] OF CHAR;
        END;
```

(continued on next page)

**Example B–8 (Cont.) Pascal SQLCA and Message Vector Declaration**

```
        SQLERRD : ARRAY [1..6] OF INTEGER;
        SQLWARN : RECORD
            SQLWARN0 : CHAR;
            SQLWARN1 : CHAR;
            SQLWARN2 : CHAR;
            SQLWARN3 : CHAR;
            SQLWARN4 : CHAR;
            SQLWARN5 : CHAR;
            SQLWARN6 : CHAR;
            SQLWARN7 : CHAR;
        END;
        SQLEXT : PACKED ARRAY [1..8] OF CHAR;
 END;
VAR
 RDB$DBHANDLE : [HIDDEN] INTEGER;
 SQLCA : [HIDDEN] SQL$SQLCA_REC;
```

OpenVMS VAX OpenVMS Alpha

Example B–9 shows the PL/I SQLCA and message vector declaration.

**Example B–9 PL/I SQLCA and Message Vector Declaration**

```
DCL 1 SQLCA  STATIC ,
     2 SQLCAID character(8) INITIAL('SQLCA   '),
     2 SQLCABC fixed binary(31) INITIAL(128),
     2 SQLCODE fixed binary(31),
     2 SQLERRM ,
      3 SQLERRML fixed binary(15) INITIAL(0),
      3 SQLERRMC character(70),
     2 SQLERRD (1:6) fixed binary(31),
     2 SQLWARN ,
      3 SQLWARN0 character(1),
      3 SQLWARN1 character(1),
      3 SQLWARN2 character(1),
      3 SQLWARN3 character(1),
      3 SQLWARN4 character(1),
      3 SQLWARN5 character(1),
      3 SQLWARN6 character(1),
      3 SQLWARN7 character(1),
     2 SQLEXT character(8);
```

(continued on next page)

**Example B–9 (Cont.) PL/I SQLCA and Message Vector Declaration**

```
DCL 1 Rdb$MESSAGE_VECTOR EXTERNAL,
     2 Rdb$LU_NUM_ARGUMENTS FIXED BINARY(31),
     2 Rdb$LU_STATUS FIXED BINARY(31),
     2 Rdb$ALU_ARGUMENTS (18),
      3 Rdb$LU_ARGUMENTS FIXED BINARY (31);
```

♦

# C
# SQLSTATE

SQL defines a set of status parameters that can be part of the parameter list for a procedure definition in a nonstored module. They are SQLSTATE, SQLCODE, and SQLCA. An SQL procedure is required to contain at least one of these status parameters in its parameter list. All status parameters are implicitly output parameters.

The purpose of these status parameters is to return the status of each SQL statement that is executed. Each status parameter gives information that allows you to determine whether the statement completed execution or an exception has occurred. These status parameters differ in the amount of diagnostic information they supply, when an exception occurs as follows:

- SQLCODE—This is the original SQL error handling mechanism. It is an integer value. SQLCODE differentiates among errors (negative numbers), warnings (positive numbers), succesful completion (0), and a special code of 100, which means no data. SQLCODE is a deprecated feature of the ANSI/ISO SQL standard.
- SQLCA—This is an extension of the SQLCODE error handling mechanism. It contains other context information that supplements the SQLCODE value. SQLCA is not part of the ANSI/ISO SQL standard. However, many databases such as DB2 and ORACLE RDBMS have defined proprietary semantics and syntax to implement it.
- SQLSTATE—This is the error handling mechanism for the ANSI/ISO SQL standard. The SQLSTATE value is a character string that is associated with diagnostic information. To use the SQLSTATE status parameter, you must specify the SQL92 dialect and compile your module using Oracle Rdb Version 6.0 or a higher version.

This appendix covers the following SQLSTATE topics:

- Definition of the SQLSTATE status parameter
- Use of the SQLSTATE status parameter

## C.1 Definition of the SQLSTATE Status Parameter

The value returned in an SQLSTATE status parameter is a string of five characters. It comprises a two-character class value followed by a three-character subclass value. Each class value corresponds to an execution condition such as success, connection exception, or data exception. Each subclass corresponds to a subset of its execution condition. For example, connection exceptions are differentiated by “connection name in use”, “connection not open”, and “connection failure” categories. A subclass of 000 means there is no subcondition.

Table C–1 shows the SQLSTATE values that SQL has defined with its corresponding execution condition. The SQLSTATE classes beginning with either the characters *R* or *S* are Oracle Rdb-specific SQLSTATE values.

**Table C–1 SQLSTATE Status Parameter Values—Sorted by SQLSTATE Class and Subclass**

| Class /Subclass | Condition | Subcondition |
|---|---|---|
| 00000 | Successful completion[1] | *No subcondition* |
| 01000 | Warning[1] | *No subcondition* |
| 01003 | | Null value eliminated in aggregate function |
| 01004 | | String data, right truncation |
| 02000 | No data | *No subcondition* |
| 08002 | Connection exception | Connection name in use |
| 08003 | | Connection not open |
| 08006 | | Connection failure |
| 21000 | Singleton select returned more than one value | *No subcondition* |

[1]For dialects other than SQL92, an SQLSTATE value of 00000 is both a warning and a success status. For the SQL92 dialect, SQLSTATE values beginning with 01 are warnings.

(continued on next page)

**Table C–1 (Cont.) SQLSTATE Status Parameter Values—Sorted by SQLSTATE Class and Subclass**

| Class /Subclass | Condition | Subcondition |
|---|---|---|
| 22001 | Data exception | String data, right truncation |
| 22002 | | Null value, no indicator parameter |
| 22003 | | Numeric value out of range |
| 22007 | | Invalid date-time format |
| 22012 | | Division by zero |
| 22018 | | Invalid character value for cast |
| 22022 | | Indicator overflow |
| 22023 | | Invalid parameter value |
| 22024 | | C string not terminated |
| 22025 | | Invalid escape sequence |
| 23000 | Integrity constraint violation | *No subcondition* |
| 24000 | Invalid cursor state | *No subcondition* |
| 25000 | Invalid transaction state | *No subcondition* |
| 26000 | Invalid SQL statement identifier | *No subcondition* |
| 37000 | Syntax error or access rule violation in dynamic SQL statement | *No subcondition* |
| 42000 | Syntax error or access rule violation | *No subcondition* |
| 44000 | With check option violation | *No subcondition* |
| R1001 | Lock error exception | Deadlock encountered |
| R1002 | | Lock conflict |
| R2000 | Duplicate on index | *No subcondition* |
| R3000 | Trigger forced an ERROR statement | *No subcondition* |
| R4000 | Distributed transaction identification error | *No subcondition* |
| R5000 | Attempted to update a read-only table | *No subcondition* |
| R6000 | Metadata no longer available | *No subcondition* |

(continued on next page)

**Table C–1 (Cont.) SQLSTATE Status Parameter Values—Sorted by SQLSTATE Class and Subclass**

| Class /Subclass | Condition | Subcondition |
|---|---|---|
| R7000 | Table in request not reserved in transaction | *No subcondition* |
| RR000 | Oracle Rdb returned an error | *No subcondition* |
| S0000 | No implicit transaction | *No subcondition* |
| S1001 | Context exception | Bad version in context structure |
| S1002 | | Bad type in context structure |
| S1003 | | Bad length in context structure |
| S2000 | Row containing list deleted | *No subcondition* |
| S3000 | Invoker was not the definer | *No subcondition* |
| S4001 | Alias exception | Alias unknown |
| S4002 | | Alias already declared |
| S7000 | Base system does not support feature being used | *No subcondition* |
| S5001 | Negative length and scale for column | Negative length specified for column |
| S5002 | | Negative scale specified for column |
| S6000 | Case not found; WHEN or ELSE not specified | *No subcondition* |

## C.2 Use of the SQLSTATE Status Parameter

Table C–1 shows the SQLSTATE classes 00, 01, and 02 as completion conditions of success, warning, and no data respectively. All other classes define exception conditions.

When using embedded SQL, the embedded exception declaration defines the following categories of exceptions:

- NOT FOUND: SQLSTATE class = 02
- SQLWARNING: SQLSTATE class = 01
- SQLEXCEPTION: SQLSTATE class > 02
- SQLERROR: SQLEXCEPTION or SQLWARNING

Example C–1 shows how to declare SQLSTATE as a parameter in a C program and how to evaluate the SQLSTATE value using the string compare function. When you declare SQLSTATE in a C program, you must type SQLSTATE in all uppercase characters.

**Example C–1 Declaring SQLSTATE in a C Program**

```
char SQLSTATE[6];
long SQLCODE;

main()
{
     EXEC SQL SELECT T_INT INTO :c1 FROM FOUR_TYPES
                WHERE T_DECIMAL = 4.1;
     printf ("SQLCODE should be < 0; its value is %ld\n", SQLCODE);
     printf ("SQLSTATE should be ’22002’; its value is %s\n", SQLSTATE);
     if (SQLCODE >= 0 || strncmp (SQLSTATE, "22002", 5) != 0)
       flag = 0;
}
```

You can use the GET DIAGNOSTICS statement to return the SQLSTATE information to your program. For more information, see the GET DIAGNOSTICS Statement.

D

# The SQL Dynamic Descriptor Areas (SQLDA and SQLDA2)

An SQL Descriptor Area (SQLDA) is a collection of parameters used only in dynamic SQL programs. SQL provides two descriptor areas: SQLDA and SQLDA2. Sections D.6 through D.6.2 include information specific to the SQLDA2.

**Dynamic SQL** lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, such dynamically executed SQL statements are not necessarily part of a program's source code, but can be generated while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program will need to process.

To use an SQLDA, host languages must support pointer variables that provide indirect access to storage by storing the address of data instead of directly storing data in the variable. The languages supported by the SQL precompiler that also support pointer variables are PL/I, C, BASIC, and Ada. Any other language that supports pointer variables can use an SQLDA, but must call SQL module procedures containing SQL statements instead of embedding the SQL statements directly in source code.

## D.1 Purpose of the SQLDA

The SQLDA provides information about dynamic SQL statements to the program and information about memory allocated by the program to SQL. Specifically, SQL and host language programs use the SQLDA for the following purposes:

- SQL uses the SQLDA as a place to *write* information about parameter markers and select list items in a prepared statement. SQL writes information about the number and data types of input and output parameter markers and select list items to the SQLDA when it processes PREPARE . . . SELECT LIST INTO statements or DESCRIBE statements.

**Parameter markers** are question marks (?) that denote parameters in the statement string of a PREPARE statement. SQL replaces parameter markers with values in parameters or dynamic memory when it executes a dynamic SQL statement.

The DESCRIBE statement writes information about select list items in a prepared SELECT statement to the SQLDA so the host language program can allocate storage (parameters or dynamic memory) for them. The storage allocated by the program then receives values in rows of the prepared SELECT statement's result table in subsequent FETCH statements.

An SQLDA at any particular time can contain information about either input or output parameter markers or select list items, but not about both:

  - SQL writes information about select list items to the SQLDA when it executes DESCRIBE . . . SELECT LIST or PREPARE . . . SELECT LIST statements.
  - SQL writes information about parameter markers to the SQLDA when it executes DESCRIBE . . . MARKERS statements. If a prepared statement has no parameter markers, a DESCRIBE . . . MARKERS statement puts values in the SQLDA to indicate that there are no parameter markers.

- The program uses the SQLDA as a place to *read* the information SQL wrote to the SQLDA about any select list items, or input or output parameter markers in the prepared statement:
  - After either a DESCRIBE . . . SELECT LIST or DESCRIBE . . . MARKERS statement, the program reads the number and data type of select list items or parameter markers.

    The program uses that information to allocate storage (either by declaring parameters or allocating dynamic memory) for values that correspond to the parameter markers or select list items.
- The program uses the SQLDA as a place to *write* the addresses of the storage it allocated for parameter markers and select list items.
- SQL uses the SQLDA as a place to *read* information about parameter markers or select list items:
  - In OPEN statements, SQL reads the addresses of a prepared SELECT statement's parameter markers to set up a cursor for the program to process.

- In FETCH statements, SQL reads the addresses of a prepared SELECT statement's select list items so it can write the values of the row being fetched to the storage allocated by the program.
- In EXECUTE statements, SQL reads the addresses of parameter markers of any prepared statement other than a SELECT statement.

The OPEN and FETCH statements used to read information from the SQLDA are not themselves dynamic statements used in a PREPARE statement, nor is a DECLARE CURSOR statement that declares the cursor named in the OPEN and FETCH statements. Although these statements *use* prepared statements, they are among the SQL statements that cannot themselves *be* prepared statements. See the PREPARE Statement for a list of statements that cannot be dynamically executed.

## D.2 How SQL and Programs Use the SQLDA

The specific sequence of operations that uses the SQLDA depends on whether a program can accept dynamically generated SELECT statements only, non-SELECT statements only, or both. The following sequence describes in general the steps a program follows in using the SQLDA. For specific examples, see the chapter on using dynamic SQL in the *Oracle Rdb7 Guide to SQL Programming* and the sample programs created during installation of Oracle Rdb in the Samples directory.

1. The program uses the embedded SQL statement INCLUDE SQLDA to automatically declare an SQLDA. In addition, the program must allocate memory for the SQLDA and set the value of one of its fields, SQLN. The value of SQLN specifies the maximum number of parameter markers or select list items about which information can be stored in the SQLDA.

   Programs can use more than one SQLDA but must explicitly declare additional SQLDA structures with names other than SQLDA. Declaring two SQLDAs can be useful for dynamic SQL programs that can accept both SELECT and non-SELECT statements. One SQLDA stores information about parameter markers and another stores information about select list items. (An alternative to declaring multiple SQLDA structures in such programs is to issue additional DESCRIBE . . . SELECT LIST statements after the program finishes with parameter marker information in the SQLDA.)

   Declaration and allocation of SQLDAs need to be done only once. The remaining steps repeat as many times as the program has dynamic SQL statements to process.

2. SQL writes the number and data types of any select list items (for a DESCRIBE . . . SELECT LIST statement) or parameter markers (for a DESCRIBE . . . MARKERS statement) of a prepared statement into the SQLDA. SQL puts the number of select list items or parameter markers in the SQLD field of the SQLDA, and stores codes denoting their data types in the SQLTYPE fields.
3. If the program needs to determine if a particular prepared statement is a SELECT statement, it reads the value of the second element of the SQLCA.SQLERRD array after a DESCRIBE . . . SELECT LIST statement. If the value is one, the prepared statement is a SELECT statement and the program needs to allocate storage for rows generated during subsequent FETCH statements.
4. When you use parameter markers in SQL statements, you should not make any assumptions about the data types of the parameters. SQL may convert the parameter to a data type that is more appropriate to a particular operation. For example, when you use a parameter marker as one value expression in a LIKE predicate, SQL returns a data type of VARCHAR for that parameter even though the other value expression has a data type of CHAR. The STARTING WITH predicate and the CONTAINING predicate treat parameter markers in the same way. You can override the VARCHAR data type in such predicates by explicitly setting the SQLTYPE field of the SQLDA to CHAR.
5. The program reads information about the number, data type, and length of any select list items (after a DESCRIBE . . . SELECT LIST statement) or parameter markers (after a DESCRIBE . . . MARKERS statement) from the SQLDA. The program then allocates storage (parameters or dynamic memory) for each of the select list items or parameters, and writes the addresses for that storage to the SQLDA. The program puts the addresses into the SQLDATA fields of the SQLDA.

   If SQL uses a data type for the parameter marker or select list item that is not supported by the programming language, the program must convert the SQLTYPE and SQLLEN fields to an appropriate data type and length. The program changes the values of SQLTYPE and SQLLEN that SQL returns from the DESCRIBE statement to a data type and length that both SQL and the host language support.
6. The program supplies values that will be substituted for parameter markers and writes those values to the storage allocated for them.

7. SQL reads information about parameter markers from the SQLDA:

   – If the prepared statement is a prepared SELECT statement, SQL reads the addresses of any parameter markers for that prepared SELECT statement when it executes an OPEN statement that refers to the SQLDA.

   – If the statement is any other prepared statement, SQL reads the addresses of parameter markers for that statement when it executes an EXECUTE statement that refers to the SQLDA.

   SQL uses the addresses of parameter markers to retrieve the values in storage (supplied by the program) and to substitute them for parameter markers in the prepared statement.

8. Finally, for prepared SELECT statements only, SQL reads the addresses of select list items when it executes a FETCH statement that refers to the SQLDA. SQL uses the information to write the values from the row of the result table to memory.

## D.3 Declaring the SQLDA

Programs can declare the SQLDA in the following ways:

- By using the INCLUDE SQLDA statement embedded in Ada, C, or PL/I programs to be precompiled. The INCLUDE SQLDA statement automatically inserts a declaration of an SQLDA structure, called SQLDA, in the program when it precompiles the program.
- In precompiled Ada programs, by specifying the SQLDA_ACCESS type in the SQL definition package. Specifying SQLDA_ACCESS offers an advantage over an embedded INCLUDE SQLDA statement because you can use it in more than one declaration to declare multiple SQLDA structures.
- In precompiled C programs and C host language programs, you can use the sql_sqlda.h header file. The following example shows how to include the file in a C program:

  ```
  #include <sql_sqlda.h>
  ```

  The sql_sqlda.h header file includes typedef statements for the SQLDA structure defining the SQL_T_SQLDA (or the SQL_T_SQLDA2) data type. In addition, it defines the SQL_T_SQLDA_FULL (or SQL_T_SQLDA2_FULL) data type as a superset to the definition of the SQLDA structure. The SQL_T_SQLDA_FULL data type is identical in layout to the SQL_T_SQLDA data type except that it contains additional unions with additional fields that SQL uses when describing CALL statements.

For additional information on declaring SQLDA structures, see the *Oracle Rdb7 Guide to SQL Programming*.

- By explicitly declaring the SQLDA in programs written in host languages that support pointer variables. Such host languages can then take advantage of dynamic SQL even though the SQL precompiler does not support them. Instead of embedding SQL statements directly in the host language source code, languages unsupported by the precompiler must call SQL module language procedures that contain SQL statements to use dynamic SQL. See Chapter 3 for more information about the SQL module language.

  Programs that explicitly declare SQLDA structures (whether or not they have precompiler support) supply a name for the SQLDA structure, which can be SQLDA or any other valid name. Declaring two SQLDAs can be useful for dynamic SQL programs that can accept both SELECT and non-SELECT statements. One SQLDA stores information about parameter markers and another stores information about select list items.

An SQLDA always includes four fields, and may sometimes include a fifth field. The fifth field, SQLVAR, is a repeating field. For languages other than C, it comprises five parameters that describe individual select list items or parameter markers of a prepared statement. For C, it comprises six parameters.

The following examples show declarations of the SQLDA for different host languages. For PL/I, C, and Ada, the examples show the declaration SQL inserts when it processes a program that contains the INCLUDE SQLDA statement. For BASIC, the example shows the format a program should use when it declares the SQLDA explicitly.

These sample declarations all use the name SQLDA as the name for the SQLDA structure, but programs can use any valid name.

OpenVMS VAX OpenVMS Alpha

Example D–1 shows the declaration that SQL inserts when it processes a program that contains the INCLUDE SQLDA statement.

**Example D–1 Declaration of the SQLDA in Ada**

```
type SQLNAME_REC is
    record
        NAME_LEN : standard.short_integer;
        NAME_STR : standard.string (1..30);
    end record;

type SQLVAR_REC is
    record
        SQLTYPE : standard.short_integer;
        SQLLEN : standard.short_integer;
        SQLDATA : system.address;
        SQLIND : system.address;
        SQLNAME : sqlname_rec;
    end record;
type SQLVAR_ARRAY is array (1..255) of sqlvar_rec;

type SQLDA_RECORD;
type SQLDA_ACCESS is access SQLDA_RECORD;
type SQLDA_RECORD is
    record
        SQLDAID : standard.string (1..8) := 'SQLDA   ';
        SQLDABC : standard.integer;
        SQLN : standard.short_integer;
        SQLD : standard.short_integer;
        SQLVAR : sqlvar_array;
    end record;
```

♦

OpenVMS VAX OpenVMS Alpha

Example D–2 shows the format that BASIC programs should use when they explicitly declare the SQLDA.

**Example D–2 Declaration of the SQLDA in BASIC**

```
RECORD  SQLDA_REC
   string SQLDAID = 8
   long SQLDABC
   word SQLN              ! Program must explicitly
   word SQLD              ! set SQLN equal to the number
   GROUP SQLVAR(100)      ! of occurrences of SQLVAR
      word SQLTYPE
      word SQLLEN
      long SQLDATA
      long SQLIND
```

(continued on next page)

**Example D–2 (Cont.) Declaration of the SQLDA in BASIC**

```
      GROUP SQLNAME
         word SQLNAME
         string SQLNAMEC = 30
      END GROUP SQLNAME
   END GROUP SQLVAR
END RECORD SQLDA_REC

DECLARE SQLDA_REC SQLDA
♦
```

Example D–3 shows the declaration that SQL inserts when it processes a C program that contains the INCLUDE SQLDA statement.

**Example D–3 Declaration of the SQLDA in C**

```
struct SQLDA_STRUCT {
          char SQLDAID[8];
          int SQLDABC;
          short SQLN;
          short SQLD;
          struct SQLVAR_STRUCT {
            short SQLTYPE;
            short SQLLEN;
            char *SQLDATA;
            short *SQLIND;
            short SQLNAME_LEN;
            char SQLNAME[30];
               } SQLVAR[1];
          } *SQLDA;
```

OpenVMS VAX OpenVMS Alpha Example D–4 shows the declaration that SQL inserts when it processes a PL/I program that contains the INCLUDE SQLDA statement.

**Example D–4 Declaration of the SQLDA in PL/I**

```
/*
    EXEC SQL INCLUDE SQLDA;
*/
DCL 1 SQLDA BASED ( SQLDAPTR ),
     2 SQLDAID  CHAR(8),
     2 SQLDABC  BIN FIXED(31),
     2 SQLN  BIN FIXED(15),
     2 SQLD  BIN FIXED(15),
     2 SQLVAR  (SQLSIZE REFER(SQLN)),
      3 SQLTYPE  BIN FIXED(15),
      3 SQLLEN  BIN FIXED(15),
      3 SQLDATA  PTR,
      3 SQLIND  PTR,
      3 SQLNAME  CHAR(30) VAR;
DCL SQLSIZE BIN FIXED;
DCL SQLDAPTR PTR;
```

♦

## D.4 Description of Fields in the SQLDA

Table D–1 describes the different fields of the SQLDA and the ways SQL uses the fields. Remember that the SQLDA, at any particular time, can contain information about either select list items or parameter markers, but not both.

**Table D–1 Fields in the SQLDA**

| Field Name | Meaning of the Field | Set by | Used by |
|---|---|---|---|
| SQLDAID | Character string field whose value is always the character string "SQLDA". | SQL | Not used. |
| SQLDABC | The length in bytes of the SQLDA, which is a function of SQLN (SQLDABC = 16 + (44 * SQLN)). | SQL | Not used. |
| SQLN | The total number of occurrences of the SQLVAR group field (the value must equal or exceed the value in SQLD, or the DESCRIBE statement). Generates a run-time error. | Program | SQL to determine if a program allocated enough storage for the SQLDA. |

(continued on next page)

**Table D–1 (Cont.) Fields in the SQLDA**

| Field Name | Meaning of the Field | Set by | Used by |
|---|---|---|---|
| SQLD | Number of select list items (if DESCRIBE . . . SELECT LIST) or parameter markers (if DESCRIBE . . . MARKERS) in prepared statement (if none, the value is 0). | SQL | Program to determine how many input or output parameters for which to allocate storage. |
| SQLVAR | A repeating group field, each occurrence of which describes a select list item or parameter marker (not used if the value of SQLD is 0). | No value | See descriptions of subfields in the following entries. |

**SQLVAR Subfields (Each Occurs Once for Each Select List Item or Parameter Marker)**

| Field Name | Meaning of the Field | Set by | Used by |
|---|---|---|---|
| SQLTYPE | A subfield of SQLVAR whose value indicates the data type of the select list item or parameter marker (see Table D–2). | SQL | Program to allocate storage with the appropriate data type for the parameter. |
| SQLLEN | A subfield of SQLVAR whose value indicates the length in bytes of the select list item or parameter marker.<br><br>For fixed-length data types (TINYINT, SMALLINT, INTEGER, QUADWORD, and DECIMAL), SQLLEN is split in half.<br><br>For TINYINT, SMALLINT, INTEGER, and QUADWORD, the low-order byte of SQLLEN indicates the length, and the high-order byte indicates the scale (the number of digits to the right of the decimal point). | SQL unless program resets, except DECIMAL or H_FLOAT, which can only be set by user | Program to allocate storage with the appropriate size for the select list item or parameter marker. |

(continued on next page)

**Table D–1 (Cont.) Fields in the SQLDA**

| | **SQLVAR Subfields (Each Occurs Once for Each Select List Item or Parameter Marker)** | | |
|---|---|---|---|
| **Field Name** | **Meaning of the Field** | **Set by** | **Used by** |
| | For DECIMAL, the low-order byte indicates the precision, and the high-order byte indicates the scale. However, the SQLLEN for a DECIMAL data type can be set only by the user; it is not returned by SQL on a DESCRIBE statement.<br><br>List cursors cannot return data in data types that require a scale factor.<br><br>For floating-point data types, the SQLLEN shows the length of the field in bytes so that SQLLEN == 4 indicates the REAL data type and F_FLOAT, SQLLEN == 8 indicates DOUBLE PRECISION (G_FLOAT), and SQLLEN == 16 indicates the H_FLOAT data type. | | |
| SQLDATA | A subfield of SQLVAR whose value is the address of the storage allocated for the select list item or parameter marker. | Program | SQL:<br>• In EXECUTE and OPEN statements, to retrieve a value stored by the program and substitute it for a parameter marker in the prepared statement.<br>• In FETCH statements, to store a value from a result table. |

(continued on next page)

**Table D–1 (Cont.) Fields in the SQLDA**

| | SQLVAR Subfields (Each Occurs Once for Each Select List Item or Parameter Marker) | | |
|---|---|---|---|
| **Field Name** | **Meaning of the Field** | **Set by** | **Used by** |
| SQLIND | A subfield of SQLVAR whose value is the address of the indicator variable, a word (16 bits) in size (if program does not set SQLIND, the value is 0). | Program | Program or SQL:<br>• In FETCH statements, by SQL, to store the value for an indicator variable associated with a select list item.<br>• After FETCH statements, by the program. to retrieve the value of a select list item's associated indicator variable.<br>• In EXECUTE and OPEN statements, by SQL, to retrieve the value of a parameter marker's associated indicator variable. |
| SQLNAME[1] | A varying character string subfield of SQLVAR whose value is:<br>For select list items, the name of the column in the select list of the prepared SELECT statement.<br>For parameter markers, the name of the column to which a parameter marker is assigned (in INSERT or UPDATE statements) or compared (in basic predicates).<br>If the select list item, assignment, or comparison involves an arithmetic expression or predicates other than basic predicates; SQL does not assign a value to SQLNAME. | SQL | The program, optionally, to find out the name of the column associated with a select list item or parameter marker. |

[1]The SQLNAME field has a maximum length of 30 bytes. SQL allows column names to be up to 31 characters. Column names of 31 characters are truncated to 30 characters when SQL stores them in SQLNAME.

Table D–2 shows the numeric and literal values for the SQLTYPE subfield of SQLVAR and the meaning of those values.

**Table D–2 Codes for SQLTYPE Field of SQLDA and SQLDA2**

| Numeric Value | Literal Value | Data Type |
|---|---|---|
| 449 | SQLDA_VARCHAR | VARCHAR[1] |
| 453 | SQLDA_CHAR | CHAR |
| 481 | SQLDA_FLOAT | FLOAT |
| 485 | SQLDA_DECIMAL | DECIMAL |
| 497 | SQLDA_INTEGER | INTEGER |
| 501 | SQLDA_SMALLINT | SMALLINT |
| 503 | SQLDA_DATE | DATE VMS |
| 505 | SQLDA_QUADWORD | QUADWORD |
| 507 | SQLDA_ASCIZ | ASCIZ[2] |
| 509 | SQLDA_SEGSTRING | LIST OF VARBYTE |
| 515 | SQLDA_TINYINT | TINYINT |
| 516 | SQLDA_VARBYTE | VARBYTE[3,4] |
| 519 | SQLDA2_DATETIME | Date-time (ANSI) |
| 521 | SQLDA2_INTERVAL | INTERVAL |

[1]For the SQLDA2 structure, VARCHAR has a longword length prefix.

[2]The SQLTYPE code for ASCIZ is never returned in the SQLDA by a DESCRIBE statement, but it can be used to override the data type that is returned.

[3]This data type value is only valid for fetches of list elements.

[4]This data type does not allow null values.

SQL provides a file that contains the declarations of all the SQLTYPE literal values. Table B–2 shows how to include this file in precompiled SQL and module language programs.

There is some confusion over the use of ASCII and ASCIZ in dynamic SQL and C programs. When a CHAR data type is written to the database using INSERT or UPDATE, the string is not padded with blank spaces. It contains a null-terminated character, which makes it difficult to access the data.

SQL does not know what the host language is when using dynamic SQL; it returns the data type of the field as in the DESCRIBE statement, (CHAR(n)), and not the data type of the user's host variable. The interpretation of CHAR(n) being ASCIZ is for host variables and not database variables.

If you change the SQLDA's SQLTYPE from CHAR to ASCIZ and increase SQLLEN by 1, no truncation occurs and the CHAR STRING fields will be padded with blank spaces accordingly (where incrementing SQLLEN by 1 accounts for the null terminator).

**Note**

SQL sets the value of SQLTYPE during the DESCRIBE statement. However, your application program can change the value of SQLTYPE to that of another data type.

For example, SQL does not support the DECIMAL data type in database columns. This means that SQL will never return the code for the DECIMAL data type in the SQLTYPE field in the SQLDA. However, programs can set the code to that for DECIMAL, and SQL will convert data from databases to DECIMAL, and data from DECIMAL parameters in the program to the data type in the database.

However, SQL assumes that program parameters will correspond to the data type indicated by the SQLTYPE code. If they do not, SQL may generate unpredictable results.

## D.5 Parameters Associated with the SQLDA: SQLSIZE and SQLDAPTR

In addition to the declaration of the SQLDA itself, SQL declares two related parameters: SQLSIZE and SQLDAPTR. These parameters can only be used in PL/I programs. The PL/I program uses both parameters when it dynamically allocates storage for the SQLDA before a DESCRIBE or PREPARE . . . SELECT LIST INTO statement. Your program must:

- Assign a value to SQLSIZE and then assign the same value to SQLN. Because the declaration of the SQLDA refers both to SQLSIZE and SQLN, the program uses that value when it allocates memory for the SQLDA.
- Dynamically allocate memory for the SQLDA based on the value assigned to SQLN, and assign the address for memory used by the SQLDA into SQLDAPTR.

The following program fragment shows how a PL/I program uses SQLSIZE and SQLDAPTR to allocate storage for the SQLDA:

```
/* Declare SQL Descriptor Area: */
EXEC SQL INCLUDE SQLDA;
/* Allocate memory for the SQLDA and
 * set the value of its SQLN field:
 */
SQLSIZE = 10;
ALLOCATE SQLDA SET (SQLDAPTR);
SQLN = 10;
```

## D.6 Purpose of the SQLDA2

SQL provides an extended version of the SQLDA, called the SQLDA2, which supports additional fields and field sizes.

You can use either the SQLDA or SQLDA2 in any dynamic SQL statement that calls for a descriptor area. SQL assumes default values for SQLDA2 fields and field sizes if you use an SQLDA structure to provide input parameters for an application; however, SQL issues an error message if the application cannot represent resulting values.

Use the SQLDA2 instead of the SQLDA when any of the following applies to the parameter markers or select list items:

- The length of the column name is greater than 30 octets. (An octet is 8 bits.)
- The data type of the column is DATE, DATE VMS, DATE ANSI, TIME, TIMESTAMP, or any of the interval data types.
- The data type is CHAR, CHAR VARYING, CHARACTER, CHARACTER VARYING, VARCHAR, or LONG VARCHAR, and any of the following is true:
  - The character set is not the default 8-bit character set.
  - The maximum length in octets exceeds 32,767.

You can examine the SQLDA2 after SQL fills in the items on a PREPARE statement. Oracle Rdb recommends this rather than setting the fields yourself.

Use one of the following methods to extract the data for your own use:

- The CAST function to convert the data to TEXT before using it
- The EXTRACT function to extract individual fields so you can format it
- The CAST function to convert to DATE VMS so that you can use OpenVMS system services

The ANSI/ISO SQL standard specifies that the data is always returned to the application program as CHAR data.

### D.6.1 Declaring the SQLDA2

Programs can declare the SQLDA2 in the same way as they declare an SQLDA, described in Section D.3.

To indicate to SQL that the structure is an SQLDA2 instead of an SQLDA, your program must set the SQLDAID field to be the character string containing the word SQLDA2 followed by two spaces.

The following examples show declarations of the SQLDA2 for different host languages. For PL/I, C, and Ada, the examples show the declaration SQL inserts when it processes a program that contains the INCLUDE SQLDA statement. For other languages, the examples show the format that programs should use when they explicitly declare the SQLDA.

OpenVMS VAX OpenVMS Alpha Example D–5 shows the declaration that SQL inserts when it processes an Ada program that contains the INCLUDE SQLDA2 statement. In this example, *N* stands for the maximum number of occurrences of SQLVAR2.

**Example D–5 Declaration of the SQLDA2 in Ada**

```
type SQLNAME_REC is
    record
        NAME_LEN : standard.short_integer;
        NAME_STR : standard.string (1..128);
    end record;
type SQLVAR_REC is
    record
        SQLTYPE : standard.short_integer;
        SQLLEN : standard.integer;
        SQLDATA : system.address;
        SQLIND : system.address;
        SQLCHRONO_SCALE: standard.integer;
        SQL_CHRONO_PRECISION: standard.integer;
        SQLNAME : sqlname_rec;
        SQLCHAR_SET_NAME : standard.string(1..128);
        SQLCHAR_SET_SCHEMA : standard.string(1..128);
        SQLCHAR_SET_CATALOG : standard.string(1..128);
    end record;
type SQLVAR_ARRAY is array (1..N) of sqlvar_rec;

type SQLDA_RECORD;
type SQLDA_ACCESS is access SQLDA_RECORD;
```

(continued on next page)

**Example D–5 (Cont.) Declaration of the SQLDA2 in Ada**

```
type SQLDA_RECORD is
    record
        SQLDAID : standard.string (1..8) := 'SQLDA2  ';
        SQLDABC : standard.integer;
        SQLN : standard.short_integer;
        SQLD : standard.short_integer;
        SQLVAR : sqlvar_array;
    end record;
```

♦

OpenVMS VAX OpenVMS Alpha Example D–6 shows the format that BASIC programs should use when they explicitly declare the SQLDA2.

**Example D–6 Declaration of the SQLDA2 in BASIC**

```
RECORD  SQLDA_REC
   string SQLDAID = 8     ! Value must be "SQLDA2  ".
   long SQLDABC
   word SQLN              ! Program must explicitly
   word SQLD              ! set SQLN equal to the number
   GROUP SQLVAR(N)        ! of occurrences of SQLVAR.
      word SQLTYPE
      long SQLLEN
      long SQLOCTET_LEN
      long SQLDATA
      long SQLIND
      long SQLCHRONO_SCALE
      long SQLCHRONO_PRECISION
      GROUP SQLNAME
         word SQLNAME
         string SQLNAMEC = 128
      END GROUP SQLNAME
      string SQLCHAR_SET_NAME = 128
      string SQLCHAR_SET_SCHEMA = 128
      string SQLCHAR_SET_CATALOG = 128
   END GROUP SQLVAR
END RECORD SQLDA_REC

DECLARE SQLDA_REC SQLDA2
```

♦

Example D–7 shows the declaration that SQL inserts when it processes a C program that contains the INCLUDE SQLDA2 statement.

**Example D–7 Declaration of the SQLDA2 in C**

```
struct SQLDA_STRUCT {
          char SQLDAID[8];  /*Value must be "SQLDA2  "*/
          int SQLDABC;      /* ignored. */
          short SQLN;
          short SQLD;
          struct {
            short SQLTYPE;
            long  SQLLEN;
            long  SQLOCTET_LEN
            char  *SQLDATA;
            long  *SQLIND;
            long  SQLCHRONO_SCALE
            long  SQLCHRONO_PRECISION
            short SQLNAME_LEN;
            char SQLNAME[128];
            char SQLCHAR_SET_NAME[128];
            char SQLCHAR_SET_SCHEMA[128];
            char SQLCHAR_SET_CATALOG[128];
          } SQLVAR[N];         /* N is maximum number of */
        } *SQLDA;                        /* occurrences of SQLVAR. */
```

## D.6.2 Description of Fields in the SQLDA2

The SQLVAR2 field for an SQLDA2 structure comprises the following parameters that describe individual select list items or parameter markers of a prepared statement:

- Length (SQLLEN and SQLOCTET_LEN fields)

**Note**

There is a big difference between the SQLLEN fields in the SQLDA and the SQLDA2. In the SQLDA, the SQLLEN field contains the length of the field in bytes. In the SQLDA2, the SQLLEN field either contains the length of the field in characters or is a subtype field for certain data types (INTERVAL and LIST OF BYTE VARYING). This is the case when you issue the DESCRIBE statement to return information from SQL to your program.

The SQLOCTET_LEN field in the SQLDA2 is analogous to the SQLLEN field in the SQLDA. Use SQLOCTET_LEN instead of

SQLLEN to allocate dynamic memory for the SQLDATA field when using the SQLDA2.

---

- Data type (SQLTYPE)
- Scale and precision (SQLLEN or SQLCHRONO_SCALE and SQLCHRONO_PRECISION)
- Character set information (SQLCHAR_SET_NAME, SQLCHAR_SET_SCHEMA, SQLCHAR_SET_CATALOG)
- Data value (SQLDATA)
- Null indicator value (SQLIND)
- Name for resulting columns of a cursor specification (SQLNAME)

Table D–3 describes the different fields of the SQLDA2 and the ways in which SQL uses the fields when passing them to dynamic SQL. Remember that the SQLDA2 at any particular time can contain information about either select list items or parameter markers, but not both.

**Table D–3 Fields in the SQLDA2**

| Field Name | Meaning of the Field | Set by | Used by |
|---|---|---|---|
| SQLDAID | Character string field whose value is always the character string "SQLDA2 " (SQLDA2 followed by two spaces). | Program | SQL to determine if the structure is an SQLDA or an SQLDA2. |
| SQLDABC | The length in bytes of the SQLDA2, which is a function of SQLN (SQLDABC = 540 * SQLN). | SQL | Not used. |
| SQLN | The total number of occurrences of the SQLVAR2 group field (the value must equal or exceed the value in SQLD or the DESCRIBE or PREPARE SELECT LIST INTO statement). Generates a run-time error. | Program | SQL to determine if program allocated enough storage for the SQLDA. |

(continued on next page)

**Table D–3 (Cont.) Fields in the SQLDA2**

| Field Name | Meaning of the Field | Set by | Used by |
|---|---|---|---|
| SQLD | Number of select list items (if DESCRIBE . . . SELECT LIST) or parameter markers (if DESCRIBE . . . MARKERS) in prepared statement (if none, the value is 0). | SQL | Program to determine how many input or output parameters for which to allocate storage. |
| SQLVAR2 | A repeating group field, each occurrence of which describes a select list item or parameter marker (not used if the value of SQLD is 0). | No value | See descriptions of subfields in following entries. |

**SQLVAR2 Subfields (Each Occurs Once for Each Select List Item or Parameter Marker):**

| Field Name | Meaning of the Field | Set by | Used by |
|---|---|---|---|
| SQLTYPE | A subfield of SQLVAR2 whose value indicates the data type of the select list item or parameter marker (see Table D–2). | SQL | Program to allocate storage with the appropriate data type for the parameter. |
| SQLLEN | A subfield of SQLVAR2 whose value indicates the length of the select list item or parameter marker.<br><br>For fixed-length data types (TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC, and DECIMAL), SQLLEN is split in half.<br><br>SQLSIZE—the low-order 16 bits<br><br>• For TINYINT, SMALLINT, INTEGER, and BIGINT, SQLSIZE and SQLOCTET_LENGTH indicate the length in bytes of the select list item or parameter marker.<br><br>• For DECIMAL, SQLSIZE indicates the precision. However, the SQLLEN for a DECIMAL data type can only be set by the user; it is not returned by SQL on a DESCRIBE statement. | | |

(continued on next page)

**Table D–3 (Cont.) Fields in the SQLDA2**

| SQLVAR2 Subfields (Each Occurs Once for Each Select List Item or Parameter Marker): | | | |
|---|---|---|---|
| **Field Name** | **Meaning of the Field** | **Set by** | **Used by** |
| | SQLSCALE—the high-order 16 bits<br>• SQLSCALE indicates the scale (the number of digits to the right of the decimal point).<br>• List cursors cannot return data in data types that require a scale factor.<br><br>For floating-point data types, SQLLEN and SQLOCTET_LEN are the size in octets of the select list item or parameter marker.<br>For DATE, DATE ANSI, DATE VMS, TIME, or TIMESTAMP, SQLLEN is the length of the date-time data type.<br>For INTERVAL data types, SQLLEN is set to one of the codes specified in Table D–4. | SQL, unless the program resets, except for DECIMAL, which can only be set by the user. | Program to allocate storage with the appropriate size for the select list item or parameter marker. |
| SQLOCTET_LEN | A subfield of SQLVAR2 whose value indicates the length in octets of the select list item or parameter marker.<br>If SQLTYPE indicates CHAR, then SQLOCTET_LEN is the maximum possible length in octets of the character string.<br>If SQLTYPE indicates CHARACTER VARYING, SQLOCTET_LEN is the maximum possible length in octets required to represent the character string, including the octets required to represent the string length. | SQL, unless the program resets. | Program or SQL. |

(continued on next page)

**Table D–3 (Cont.) Fields in the SQLDA2**

| SQLVAR2 Subfields (Each Occurs Once for Each Select List Item or Parameter Marker): | | | |
|---|---|---|---|
| **Field Name** | **Meaning of the Field** | **Set by** | **Used by** |
| | If SQLTYPE indicates a fixed-scale or floating-point numeric data type, SQLOCTET_LEN is the size in octets of the numeric select list item or parameter marker.<br><br>If SQLTYPE indicates a date-time or interval data type, then dynamic SQL ignores SQLOCTET_LEN. | | |
| SQLCHRONO_ SCALE | A longword subfield of SQLVAR2 whose value indicates the specific date-time data type of the column.<br><br>When SQLTYPE represents a date-time data type, SQLCHRONO_SCALE contains a code specified in Table D–5.<br><br>When SQLTYPE represents an interval data type, SQLCHRONO_SCALE contains the implied or specified interval leading field precision.<br><br>When SQLTYPE represents a data type that is neither date-time nor interval, SQLCHRONO_SCALE contains 0. | SQL, unless the program resets. | Program. |
| SQLCHRONO_ PRECISION | A longword subfield of SQLVAR2 whose value indicates the precision of the column represented by SQLVAR2 when that column has a date-time data type.<br><br>When SQLTYPE represents a TIME or TIMESTAMP data type, SQLCHRONO_PRECISION contains the time precision or timestamp precision.<br><br>When SQLTYPE represents an interval data type with a fractional seconds precision, SQLCHRONO_PRECISION is set to that value. Otherwise, SQLCHRONO_PRECISION is set to 0. | SQL, unless the program resets. | Program. |

(continued on next page)

**Table D–3 (Cont.) Fields in the SQLDA2**

| | **SQLVAR2 Subfields (Each Occurs Once for Each Select List Item or Parameter Marker):** | | |
|---|---|---|---|
| **Field Name** | **Meaning of the Field** | **Set by** | **Used by** |
| SQLCHAR_SET_ NAME | A 128-byte subfield of SQLVAR2 whose value is the character set name if SQLTYPE is a character string type, and spaces if SQLTYPE is any other data type. | SQL, unless the program resets. | The SQLCHAR_SET_NAME field indicates the character set name of a select list item or parameter marker if the select list item or parameter marker has a character data type. Table D–6 shows the possible values for the SQLCHAR_SET_NAME field when the SQLTYPE indicates one of the character data types. |
| SQLCHAR_SET_ SCHEMA | A 128-byte subfield of SQLVAR2 whose value is the character set of the schema name if SQLTYPE is a character string type, and spaces if SQLTYPE is any other data type. | Reserved for future use. | Reserved for future use. |
| SQLCHAR_SET_ CATALOG | A 128-byte subfield of SQLVAR2 whose value is the character set of the catalog name if SQLTYPE is a character string type, and spaces if SQLTYPE is any other data type. | Reserved for future use. | Reserved for future use. |
| SQLDATA | A subfield of SQLVAR2 whose value is the address of the storage allocated for the select list item or parameter marker. | Program. | SQL:<br>• In EXECUTE and OPEN statements, to retrieve a value stored by the program and substitute it for a parameter marker in the prepared statement.<br>• In FETCH statements, to store a value from a result table. |

(continued on next page)

**Table D–3 (Cont.) Fields in the SQLDA2**

| | | | |
|---|---|---|---|
| | **SQLVAR2 Subfields (Each Occurs Once for Each Select List Item or Parameter Marker):** | | |
| **Field Name** | **Meaning of the Field** | **Set by** | **Used by** |
| SQLIND | A subfield of SQLVAR2 whose value is the address of a longword indicator variable, a longword (32 bits) in size (if the program does not set an indicator variable, the value is 0). | Program. | Program or SQL:<br>• In FETCH statements, by SQL to store the value for an indicator variable associated with a select list item.<br>• After FETCH statements, by program to retrieve the value of a select list item's associated indicator variable.<br>• In EXECUTE and OPEN statements, by SQL to retrieve the value of a parameter marker's associated indicator variable. |
| SQLNAME | A varying character string subfield of SQLVAR2 whose value is:<br>• For select list items, the name of the column in the select list of the prepared SELECT statement.<br>• For parameter markers, the name of the column to which a parameter marker is assigned (in INSERT or UPDATE statements) or compared (in basic predicates).<br>If the select list item, assignment, or comparison involves an arithmetic expression or predicates other than basic predicates, SQL does not assign a value to SQLNAME. | SQL. | Program, optionally, to find out the name of the column associated with a select list item or parameter marker. |
| SQLNAME_LEN | A subfield of SQLVAR2 whose value is the length in octets of the column named by SQLNAME. | | |

Table D–4 shows the possible values for the SQLLEN field when the SQLTYPE indicates one of the interval data types.

**Table D–4 Codes for Interval Qualifiers in the SQLDA2**

| Code | Interval Qualifier | Interval Subtype |
|---|---|---|
| 1 | YEAR | SQLDA2_DT_YEAR |
| 2 | MONTH | SQLDA2_DT_MONTH |
| 3 | DAY | SQLDA2_DT_DAY |
| 4 | HOUR | SQLDA2_DT_HOUR |
| 5 | MINUTE | SQLDA2_DT_MINUTE |
| 6 | SECOND | SQLDA2_DT_SECOND |
| 7 | YEAR TO MONTH | SQLDA2_DT_YEAR_MONTH |
| 8 | DAY TO HOUR | SQLDA2_DT_DAY_HOUR |
| 9 | DAY TO MINUTE | SQLDA2_DT_DAY_MINUTE |
| 10 | DAY TO SECOND | SQLDA2_DT_DAY_SECOND |
| 11 | HOUR TO MINUTE | SQLDA2_DT_HOUR_MINUTE |
| 12 | HOUR TO SECOND | SQLDA2_DT_HOUR_SECOND |
| 13 | MINUTE TO SECOND | SQLDA2_DT_MINUTE_SECOND |

Table D–5 shows the possible values for the SQLCHRONO_SCALE field when SQLTYPE indicates the data type DATE, DATE ANSI, DATE VMS, TIME or TIMESTAMP.

**Table D–5 Codes for Date-Time Data Types in the SQLDA2**

| Code | Date-Time Data Type | Date-Time Subtypes |
|---|---|---|
| 1 | DATE ANSI | SQLDA2_DT_DATE |
| 2 | TIME | SQLDA2_DT_TIME |
| 3 | TIMESTAMP | SQLDA2_DT_TIMESTAMP |
| 4 | TIME WITH TIME ZONE | SQLDA2_DT_TIME_TZ |
| 5 | TIMESTAMP WITH TIME ZONE | SQLDA2_DT_TIMESTAMP_TZ |

Table D–6 shows the possible values for the SQLCHAR_SET_NAME field when the SQLTYPE indicates one of the character data types.

**Table D–6 Values for the SQLCHAR_SET_NAME Field**

| Character Set Value | Description |
|---|---|
| DEFAULT | Database default character set |
| NATIONAL | National character set |
| DEC_MCS | MCS |
| KOREAN | Korean |
| DEC_KOREAN | Korean and ASCII |
| HANYU | Hanyu |
| DEC_SICGCC | Hanyu and ASCII |
| DEC_HANYU | Hanyu, ASCII, and Supplemental Taiwanese |
| HANZI | Hanzi |
| DEC_HANZI | Hanzi and ASCII |
| KANJI | Kanji |
| DEC_KANJI | Kanji and ASCII |
| KATAKANA | Katakana |
| ISOLATINARABIC | ISO-Latin Arabic |
| ISOLATINHEBREW | ISO-Latin Hebrew |
| ISOLATINCYRILLIC | ISO-Latin Cyrillic |
| ISOLATINGREEK | ISO-Latin Greek |
| DEVANAGARI | Devanagari |
| UNSPECIFIED | The character set is unspecified. SQL does not check for compatibility of data. |

# E
# Logical Names and Configuration Parameters Used by SQL

Table E–1 lists the logical names and configuration parameters that SQL recognizes for special purposes.

**Table E–1 Summary of SQL Logical Names and Configuration Parameters**

| Logical Name | Configuration Parameter | Function |
|---|---|---|
| RDB$CHARACTER_SET | Not applicable | Specifies the database default and national character sets in addition to the session default, identifier, literal, and national character sets. Table E–2 shows the valid equivalence names for the logical name.<br><br>The logical name is used by the EXPORT and IMPORT statements and by the SQL precompiler and SQL module language to allow compatibility of most recent versions with earlier versions of Oracle Rdb. This logical name sets the attributes for the default connection.<br><br>This logical name is available only on the OpenVMS platform. This logical name is also deprecated and will not be supported in a future release. |

(continued on next page)

**Table E–1 (Cont.) Summary of SQL Logical Names and Configuration Parameters**

| Logical Name | Configuration Parameter | Function |
|---|---|---|
| RDB$LIBRARY | Not applicable | Specifies a protected library that you can use to store external routine images, such as external functions. Oracle Rdb recommends that you manage public or sensitive external routine images using a protected library that is referenced by the logical name RDB$LIBRARY. You should define RDB$LIBRARY as an executive mode logical name in the system logical name table. If the external routine image is located in the protected area, you can ensure that the desired image is used by specifying the RDB$LIBRARY logical name with an explicit file name in the LOCATION clause plus the WITH SYSTEM LOGICAL_NAME TRANSLATION clause in a CREATE FUNCTION statement. |
| RDB$ROUTINES | Not applicable | Specifies the location of an external routine image. If you do not specify a location clause in a CREATE FUNCTION or CREATE PROCEDURE statement, or if you specify the DEFAULT LOCATION clause, SQL uses the RDB$ROUTINES logical name as the default image location. |
| RDM$BIND_ABS_QUIET_POINT | RDB_BIND_ABS_QUIET_POINT | Enables quiet-point after-image journal backup operations. |
| RDM$BIND_ABW_ENABLED | RDB_BIND_ABW_ENABLED | Enables asynchronous batch-write operations. See Appendix F for information about the RDM$BIND_ABW_DISABLED logical name. |
| RDM$BIND_APF_ENABLED | RDB_BIND_APF_ENABLED | Enables asynchronous prefetch operations. See Appendix F for information about the RDM$BIND_APF_DISABLED logical name. |
| RDM$BIND_LOCK_TIMEOUT_INTERVAL | RDB_BIND_LOCK_TIMEOUT_INTERVAL | Specifies the amount of time a transaction waits for locks to be released. |
| RDM$BIND_STATS_ENABLED | RDB_BIND_STATS_ENABLED | Allows you to enable the writing of database statistics for a process. Disabling statistics is useful for static, performance-critical applications that have been previously tuned and do not need the information provided by the statistics collection package. See Appendix F for information about the RDM$BIND_STATS_DISABLED logical name. |
| RDMS$BIND_OUTLINE_MODE | RDB_BIND_OUTLINE_MODE | When multiple outlines exist for a query, this logical name is defined to select which outline to use. |

(continued on next page)

**Table E–1 (Cont.) Summary of SQL Logical Names and Configuration Parameters**

| Logical Name | Configuration Parameter | Function |
|---|---|---|
| RDMS$BIND_PRESTARTED_TXN | RDB_BIND_PRESTARTED_TXN | Defines the default setting for prestarted transactions outside an application. |
| RDMS$BIND_QG_CPU_TIMEOUT | RDB_BIND_QG_CPU_TIMEOUT | Specifies the amount of CPU time used to optimize a query for execution. |
| RDMS$BIND_QG_REC_LIMIT | RDB_BIND_QG_REC_LIMIT | Specifies the number of rows that SQL fetches before the query governor stops output. |
| RDMS$BIND_QG_TIMEOUT | RDB_BIND_QG_TIMEOUT | Specifies the number of seconds that SQL spends compiling a query before the query governor aborts that query. |
| RDMS$BIND_SEGMENTED_STRING_BUFFER | RDB_BIND_SEGMENTED_STRING_BUFFER | Allows you to reduce the overhead of I/O operations at run time when you are manipulating a segmented string. |
| RDMS$DEBUG_FLAGS | RDB_DEBUG_FLAGS | Allows you to examine database access strategies and the estimated cost of those strategies when your program runs. |
| RDMS$DIAG_FLAGS | RDB_DIAG_FLAGS | When defined to 'L', prevents the opening of a scrollable list cursor when the online format of lists is chained. |
| RDMS$RTX_SHRMEM_PAGE_CNT | RDB_RTX_SHRMEM_PAGE_CNT | Specifies the size of the shared memory area used to manipulate server site-bound, external routine parameter data and control data. |
| RDMS$USE_OLD_CONCURRENCY | RDB_USE_OLD_CONCURRENCY | Allows applications to use the isolation-level behavior that was in effect for V4.1. |
| RDMS$USE_OLD_SEGMENTED_STRING | RDB_USE_OLD_SEGMENTED_STRING | When defined to YES, the default online format for lists (segmented strings) is chained. |
| RDMS$VALIDATE_ROUTINE | RDB_VALIDATE_ROUTINE | Controls the validation of routines. |
| SQL$DATABASE | SQL_DATABASE | Specifies the database that SQL declares if you do not explicitly declare a database. |
| SQL$DISABLE_CONTEXT | SQL_DISABLE_CONTEXT | Disables the two-phase commit protocol. Useful for turning off distributed transactions when you want to run batch-update transactions. |
| SQL$EDIT | SQL_EDIT | Specifies the editor that SQL invokes when you issue the EDIT statement in interactive SQL. See the EDIT Statement for details. |

(continued on next page)

**Table E–1 (Cont.) Summary of SQL Logical Names and Configuration Parameters**

| Logical Name | Configuration Parameter | Function |
|---|---|---|
| SQL$KEEP_PREP_FILES | SQL_KEEP_PREP_FILES | On Oracle Rdb for OpenVMS VAX, causes the SQL precompiler and module processor to retain the intermediate macro (.MAR and .MOB) files. By default, these files are deleted when the precompiler or module processor completes processing your source file. SQL checks to see if the logical name is defined (as any value), and does not delete the intermediate files if it is. The .MAR files sometimes help to diagnose problems in precompiled programs and programs that call SQL modules. The .MOB files are object files representing the SQL code.<br><br>On Oracle Rdb for OpenVMS Alpha, the SQL precompiler and module processor do not generate .MAR files, even when the SQL$KEEP_PREP_FILES logical name is defined. |
| SQLINI | SQLINI | Specifies the command file that SQL executes when you invoke interactive SQL. |
| SYS$CURRENCY | Not applicable | Specifies the character that SQL substitutes for the dollar sign ($) symbol in an EDIT STRING clause of a column or domain definition. |
| SYS$DIGIT_SEP | Not applicable | Specifies the character that SQL substitutes for the comma symbol (,) in an EDIT STRING clause of a column or domain definition. |
| SYS$LANGUAGE | Not applicable | Specifies the language that SQL uses for date and time input and displays. |
| SYS$RADIX_POINT | Not applicable | Specifies the character that SQL substitutes for the decimal point symbol (.) in an EDIT STRING clause of a column or domain definition. |

Table E–2 shows the valid equivalence names for the logical name RDB$CHARACTER_SET.

**Table E–2 Valid Equivalence Names for RDB$CHARACTER_SET Logical Name**

| Character Set | Name of Character Set | Equivalence Name |
|---|---|---|
| MCS | DEC_MCS | Undefined |
| Korean and ASCII | DEC_KOREAN | DEC_HANGUL |
| Hanyu and ASCII | DEC_HANYU | DEC_HANYU |
| Hanzi and ASCII | DEC_HANZI | DEC_HANZI |
| Kanji and ASCII | DEC_KANJI | DEC_KANJI |

For more information on these and other logical names, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

# F
# Obsolete SQL Syntax

This appendix describes:

- Incompatible syntax

  Certain SQL statements that were allowed in earlier versions of SQL now have different behavior that is incompatible with earlier versions. *You must modify existing applications.*

- Deprecated syntax

  Certain SQL statements that were allowed in earlier versions of SQL will be identified (flagged) with diagnostic messages. SQL refers to such statements as deprecated features. Although these statements will process with expected behavior for this release, SQL may not support them in future versions. You should replace deprecated syntax with the new syntax in applications.

- Reserved words deprecated as identifiers

  If any of the listed reserved words is used as an identifier without double quotation marks ( " ), SQL flags the usage as being noncompliant with the ANSI/ISO standard and issues a deprecated feature message.

- Punctuation changes

  This section describes changes to punctuation marks used in SQL.

- Suppressing diagnostic messages

  This section describes how to suppress the diagnostic messages about deprecated features.

## F.1 Incompatible Syntax

The following sections describe incompatible syntax.

## F.1.1 Incompatible Syntax Containing the SCHEMA Keyword

Because one database may contain multiple schemas, the following incompatible changes apply to SQL syntax containing the SCHEMA keyword.

### F.1.1.1 CREATE SCHEMA Meaning Incompatible

Use of the CREATE SCHEMA statement to create a database is deprecated. If you use the CREATE SCHEMA statement to specify the physical attributes of a database such as the root file parameters, SQL issues the deprecated feature message and interprets the statement as it did in previous versions of SQL.

```
SQL> CREATE SCHEMA PARTS SNAPSHOT IS ENABLED;
%SQL-I-DEPR_FEATURE, Deprecated Feature: SCHEMA (meaning DATABASE)
SQL>
```

However, if you do not specify any physical attributes of a database, you must enable multischema naming to use the CREATE SCHEMA statement.

```
SQL> CREATE SCHEMA PARTS;
%SQL-F-SCHCATMULTI, Schemas and catalogs may only be referenced with
multischema enabled
```

When you enable multischema naming, the CREATE SCHEMA statement creates a new schema within the current catalog.

```
SQL> ATTACH 'ALIAS Q4 FILENAME INVENTORY MULTISCHEMA IS ON';
SQL> CREATE SCHEMA PARTS;
SQL> SHOW SCHEMAS;
Schemas in database with alias Q4
    RDB$SCHEMA
    PARTS
```

### F.1.1.2 SHOW SCHEMA Meaning Incompatible

If you use a SHOW SCHEMA statement when you are attached to a database with the multischema attribute and have multischema naming enabled, SQL shows all the schemas for the current catalog. To show a database, use the SHOW DATABASE or SHOW ALIAS statement.

If you use a SHOW SCHEMA statement when you do not have multischema enabled, SQL issues an error message.

### F.1.1.3 DROP SCHEMA Meaning Incompatible

If you use a DROP SCHEMA statement when you are attached to a database with the multischema attribute and have multischema naming enabled, SQL deletes the named schema from that database.

If you use a DROP SCHEMA statement when you do not have multischema enabled, SQL issues an error message.

If you use a DROP SCHEMA FILENAME statement, SQL interprets this as it would have in V4.0 and prior versions; it deletes the database with the named file name, and issues a deprecated feature error message.

### F.1.2 DROP TABLE Now Restricts by Default

In V4.1 and higher, the default behavior of the DROP TABLE statement is a restricted delete, not a cascading delete as in earlier versions. Only the table will be deleted. If other items (views, constraints, indexes, or triggers) refer to the specified table, the delete will fail, as shown in the following example:

```
SQL> DROP TABLE DEGREES;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-TRGEXI, relation DEGREES is referenced in trigger
COLLEGE_CODE_CASCADE_UPDATE
-RDMS-F-RELNOTDEL, relation DEGREES has not been deleted
```

If you specify the CASCADE keyword for SQL DROP TABLE statements, SQL deletes all items that refer to the table or view, then deletes the table itself. The following example shows a cascading delete:

```
SQL> DROP TABLE CASCADE JOB_HISTORY;
View CURRENT_INFO is also being dropped.
View CURRENT_JOB is also being dropped.
Constraint JOB_HISTORY_FOREIGN1 is also being dropped.
Constraint JOB_HISTORY_FOREIGN2 is also being dropped.
Constraint JOB_HISTORY_FOREIGN3 is also being dropped.
Index JH_EMPLOYEE_ID is also being dropped.
Index JOB_HISTORY_HASH is also being dropped.
VIA clause on storage map JOB_HISTORY_MAP is also being dropped.
Trigger EMPLOYEE_ID_CASCADE_DELETE is also being dropped.
```

### F.1.3 Database Handle Names Restricted to 25 Characters

The database handle name is called an alias in SQL. When sessions are enabled by the OPTIONS=(CONNECT) qualifier on the SQL precompiler command line or the CONNECT qualifier on the module language command line, the length of an alias can be no more than 25 characters. The database handle was called an authorization identifier in versions of SQL prior to V4.1.

### F.1.4 Deprecated Default Semantics of the ORDER BY Clause

In V4.1 and previous versions, SQL had the following default semantics:

- The ANSI/ISO 1989 standard provides a different direction. In future releases, SQL will assign the sort order of ASC to any key not specifically qualified with DESC.

- SQL will issue a deprecated feature warning if any sort keys inherit the DESC qualifier.

**Note**

If you do not specify ASC or DESC for the second or subsequent sort keys, SQL uses the order you specified for the preceding sort keys. If you do not specify the sorting order with the first sort key, the default order is ascending.

### F.1.5 Change to EXTERNAL NAMES IS Clause

The multischema EXTERNAL NAME IS clause has changed to the STORED NAME IS clause to avoid confusion with future SQL standards.

## F.2 Deprecated Syntax

Table F–1 lists SQL statements that have been replaced by new syntax. These statements will be allowed by SQL, but in some cases SQL flags the statement with a deprecated feature message.

**Table F–1 Deprecated Keywords for SQL**

| Deprecated Statement | New Syntax | Deprecated Feature Message? |
|---|---|---|
| ALTER SCHEMA | ALTER DATABASE | Yes |
| CREATE SCHEMA | CREATE DATABASE | Yes[1] |
| DECLARE SCHEMA — module language and precompiled SQL | DECLARE ALIAS | Yes |
| DECLARE SCHEMA — dynamic and interactive SQL | ATTACH | In interactive SQL, but not in dynamic SQL |
| DECLARE and SET TRANSACTION — CONSISTENCY LEVEL 2, 3 | ISOLATION LEVEL READ COMMITTED<br>ISOLATION LEVEL REPEATABLE READ<br>ISOLATION LEVEL SERIALIZABLE | Yes |
| DROP SCHEMA FILENAME | DROP DATABASE FILENAME | Message only in precompiled SQL and SQL module language |
| DROP SCHEMA PATHNAME | DROP DATABASE PATHNAME | Message only in precompiled SQL and SQL module language |

[1]See Section F.1 for more information.

(continued on next page)

**Table F–1 (Cont.) Deprecated Keywords for SQL**

| Deprecated Statement | New Syntax | Deprecated Feature Message? |
|---|---|---|
| DROP SCHEMA AUTHORIZATION | DROP DATABASE ALIAS | Message only in precompiled SQL and SQL module language |
| EXPORT SCHEMA FILENAME | EXPORT DATABASE FILENAME | No |
| EXPORT SCHEMA PATHNAME | EXPORT DATABASE PATHNAME | No |
| EXPORT SCHEMA AUTHORIZATION | EXPORT DATABASE ALIAS | No |
| FINISH | DISCONNECT DEFAULT | Yes, if databases are declared with DECLARE SCHEMA; otherwise, error message on nonconforming usage |
| GRANT on SCHEMA AUTHORIZATION | GRANT ON DATABASE ALIAS | Yes |
| IMPORT SCHEMA AUTHORIZATION | IMPORT DATABASE FROM filespec WITH ALIAS | Yes |
| INTEGRATE | INTEGRATE DATABASE | Yes |
| PREPARE . . . SELECT LIST | DESCRIBE . . . SELECT LIST | Yes |
| REVOKE | REVOKE ON DATABASE ALIAS | Yes |
| SET ANSI | SET DEFAULT DATE FORMAT<br>SET KEYWORD RULES<br>SET QUOTING RULES<br>SET VIEW UPDATE RULES | No |

## F.2.1 Command Line Qualifiers

Certain qualifiers in the SQL module language and precompiler command lines have been replaced. These are:

- The ANSI_AUTHORIZATION qualifier is replaced by the RIGHTS clause.
- The ANSI_DATE qualifier is replaced by the DEFAULT DATE FORMAT clause.
- The ANSI_IDENTIFIERS qualifier is replaced by the KEYWORD RULES clause.
- The ANSI_PARAMETERS qualifier is replaced by the PARAMETER COLONS clause.
- The ANSI_QUOTING qualifier is replaced by the QUOTING RULES clause.

## F.2.2 Deprecated Interactive SQL Statements

If you use the SET ANSI statement, SQL returns a deprecated feature message. This statement has been replaced by:

- The SET ANSI DATE statement is replaced by the SET DEFAULT DATE FORMAT statement. See the SET DEFAULT DATE FORMAT Statement for more information.
- The SET ANSI IDENTIFIERS statement is replaced by the SET KEYWORD RULES statement. See the SET KEYWORD RULES Statement for more information.
- The SET ANSI QUOTING statement is replaced by the SET QUOTING RULES statement. See the SET QUOTING RULES Statement for more information.

## F.2.3 Constraint Conformance to the ANSI/ISO SQL Standard

The location of the constraint name in the CREATE TABLE statement has been changed for ANSI/ISO SQL conformance. Constraint names are expected before the constraint rather than after. If you place a constraint name after the constraint, you get the following deprecated feature message:

```
SQL> CREATE TABLE TEMP2
cont> (COL1 REAL NOT NULL CONSTRAINT C7);
%SQL-I-DEPR_FEATURE, Deprecated Feature: Constraint name clause following
constraint definition
%SQL-I-DEPR_FEATURE, Deprecated Feature: Default evaluation for constraints:
DEFERRABLE
```

The default evaluation time of DEFERRABLE for constraints has been deprecated. If your dialect is SQLV40, constraints are still DEFERRABLE by default. However, you will receive the following deprecated feature message if you do not specify an evaluation time:

```
SQL> CREATE TABLE TEMP3
cont> (COL1 REAL CONSTRAINT C6 NOT NULL);
%SQL-I-DEPR_FEATURE, Deprecated Feature: Default evaluation for constraints:
DEFERRABLE
```

If your dialect is SQL92, constraints are NOT DEFERRABLE by default, and you do not receive deprecated feature messages.

### F.2.4 Obsolete Keywords

Table F–2 lists obsolete keywords and preferred substitutes for SQL statements.

**Table F–2 Obsolete SQL Keywords**

| Obsolete Keyword | Preferred Keyword |
|---|---|
| COMMIT_TIME | COMMIT TIME |
| CREATETAB | CREATE |
| DIAGNOSTIC | CONSTRAINT |
| QUADWORD | BIGINT |
| READ_ONLY | READ ONLY |
| READ_WRITE | READ WRITE |
| UNIQUE | SINGLE |
| VERB_TIME | VERB TIME |

If you use the obsolete keywords, you receive the following diagnostic message:

```
 SET TRANSACTION READ_ONLY;
                 1
%SQL-I-DEPR_FEATURE, (1) Deprecated Feature
```

## F.3 Deprecated Logical Names

The following sections describe deprecated logical names and, if applicable, the logical name replacement.

See Appendix E for more information regarding any new logical names.

### F.3.1 RDM$BIND_*n*_DISABLED Logical Names

Table F–3 shows logical names that were introduced in Oracle Rdb V6.0 and the V6.1 logical name replacements. The V6.0 logical names continue to be supported for compatibility with all previous versions, but they have been deprecated in Oracle Rdb V6.1.

**Table F–3 Logical Name Changes**

| V6.0 OpenVMS Logical Name | V6.1 OpenVMS Logical Name |
|---|---|
| RDM$BIND_ABW_DISABLED | RDM$BIND_ABW_ENABLED |
| RDM$BIND_APF_DISABLED | RDM$BIND_APF_ENABLED |
| RDM$BIND_STATS_DISABLED | RDM$BIND_STATS_ENABLED |

SQL syntax was introduced in Oracle Rdb V6.1 for these features. Oracle Rdb recommends that you use the SQL syntax for these features. See the CREATE DATABASE Statement and ALTER DATABASE Statement for more information regarding the new syntax.

### F.3.2 RDB$CHARACTER_SET Logical Name

The logical name RDB$CHARACTER_SET has been deprecated. It is used by SQL to allow compatibility for databases and applications from V4.1 and V4.0.

When you are using versions higher than V4.1 and V4.0, Oracle Rdb recommends that you use the following clauses and statements in place of the logical name:

- The DEFAULT CHARACTER SET and NATIONAL CHARACTER SET clauses in the DECLARE ALIAS statement.
- The IDENTIFIER CHARACTER SET, DEFAULT CHARACTER SET, and NATIONAL CHARACTER SET clauses of the SQL module header (Section 3.2) or the DECLARE MODULE statement.
- The SET IDENTIFIER CHARACTER SET statement, SET DEFAULT CHARACTER SET statement, and the SET NATIONAL CHARACTER SET statement for dynamic SQL.
- The IDENTIFIER CHARACTER SET, DEFAULT CHARACTER SET, and NATIONAL CHARACTER SET clauses in the CREATE DATABASE statement or the ALTER DATABASE statement.

## F.4 Reserved Words Deprecated as Identifiers

The following lists contain reserved words from the:

- ANSI/ISO 1989 SQL standard
- ANSI/ISO 1992 SQL standard
- ANSI/ISO SQL3 draft standard

If these reserved words are used as identifiers without double quotation marks ("), SQL flags their use as being noncompliant with the ANSI/ISO 1989 standard and issues a deprecated feature message.

Oracle Rdb does not recommend using reserved words as identifiers because this capability is a deprecated feature and might not be supported in future versions of SQL. However, if you must use reserved words as identifiers, then you must enclose them within double quotation marks to be compliant with the ANSI/ISO 1989 standard. SQL does not allow lowercase letters, spaces, or tab stops within the double quotation marks.

For example, if you want to use the ANSI/ISO 1989 reserved word SELECT as a table identifier, the statement would be written as follows:

```
SELECT * FROM "SELECT";
```

### F.4.1 ANSI/ISO 1989 SQL Standard Reserved Words

| | | |
|---|---|---|
| ALL | FETCH | ORDER |
| AND | FLOAT | PASCAL |
| ANY | FOR | PLI |
| AS | FOREIGN | PRECISION |
| ASC | FORTRAN | PRIMARY |
| AUTHORIZATION | FOUNT | PRIVILEGES |
| AVG | FROM | PROCEDURE |
| BEGIN | GO | PUBLIC |
| BETWEEN | GOTO | REAL |
| BY | GRANT | REFERENCES |
| CHAR | GROUP | ROLLBACK |
| CHARACTER | HAVING | SCHEMA |
| CHECK | IN | SECTION |
| CLOSE | INDICATOR | SELECT |
| COBOL | INSERT | SET |
| COMMIT | INT | SMALLINT |
| CONTINUE | INTEGER | SOME |
| COUNT | INTO | SQL |
| CREATE | IS | SQLCODE |
| CURRENT | KEY | SQLERROR |

| | | |
|---|---|---|
| CURSOR | LANGUAGE | SUM |
| DEC | LIKE | TABLE |
| DECIMAL | MAX | TO |
| DECLARE | MIN | UNION |
| DEFAULT | MODULE | UNIQUE |
| DELETE | NOT | UPDATE |
| DESC | NULL | USER |
| DISTINCT | NUMERIC | VALUES |
| DOUBLE | OF | VIEW |
| END | ON | WHENEVER |
| ESCAPE | OPEN | WHERE |
| EXEC | OPTION | WITH |
| EXISTS | OR | WORK |

### F.4.2 ANSI/ISO 1992 SQL Standard Reserved Words

In addition to the reserved words listed for the ANSI/ISO 1989 standard, the ANSI/ISO SQL standard also includes the following reserved words:

| | | |
|---|---|---|
| ABSOLUTE | ELSE | PARTIAL |
| ACTION | END-EXEC | POSITION |
| ADD | EXCEPT | PREPARE |
| ALLOCATE | EXCEPTION | PRESERVE |
| ALTER | EXECUTE | PRIOR |
| ARE | EXTERNAL | READ |
| ASSERTION | EXTRACT | RELATIVE |
| AT | FALSE | RESTRICT |
| BIT | FIRST | REVOKE |
| BIT_LENGTH | FULL | RIGHT |
| BOTH | GET | ROWS |
| CASCADE | GLOBAL | SCROLL |
| CASCADED | HOUR | SECOND |
| CASE | IDENTITY | SESSION |
| CAST | IMMEDIATE | SESSION_USER |
| CATALOG | INITIALLY | SIZE |

|  |  |  |
|---|---|---|
| CHAR_LENGTH | INNER | SPACE |
| CHARACTER_LENGTH | INPUT | SQLSTATE |
| COALESCE | INSENSITIVE | SUBSTRING |
| COLLATE | INTERSECT | SYSTEM_USER |
| COLLATION | INTERVAL | TEMPORARY |
| COLUMN | ISOLATION | THEN |
| CONNECT | JOIN | TIME |
| CONNECTION | LAST | TIMESTAMP |
| CONSTRAINT | LEADING | TIMEZONE_HOUR |
| CONSTRAINTS | LEFT | TIMEZONE_MINUTE |
| CONVERT | LEVEL | TRAILING |
| CORRESPONDING | LOCAL | TRANSACTION |
| CROSS | LOWER | TRANSLATE |
| CURRENT_DATE | MATCH | TRANSLATION |
| CURRENT_TIME | MINUTE | TRIM |
| CURRENT_TIMESTAMP | MONTH | TRUE |
| CURRENT_USER | NAMES | UNKNOWN |
| DATE | NATIONAL | UPPER |
| DAY | NATURAL | USAGE |
| DEALLOCATE | NCHAR | USING |
| DEFERRABLE | NEXT | VALUE |
| DEFERRED | NO | VARCHAR |
| DESCRIBE | NULLIF | VARYING |
| DESCRIPTOR | OCTET_LENGTH | WHEN |
| DIAGNOSTICS | ONLY | WRITE |
| DISCONNECT | OUTER | YEAR |
| DOMAIN | OUTPUT | ZONE |
| DROP | OVERLAPS |  |
|  | PAD |  |

### F.4.3 ANSI/ISO SQL3 Reserved Words

In addition to the reserved words listed for the ANSI/ISO 1989 standard and the ANSI/ISO SQL standard, the ANSI/ISO SQL3 draft standard includes the following reserved words as of February, 1993. This list is a work-in-progress and is undergoing constant change.

| | | |
|---|---|---|
| ACTOR | LESS | RETURNS |
| AFTER | LIMIT | ROLE |
| ALIAS | LIST | ROUTINE |
| ASYNC | LOOP | ROW |
| BEFORE | MODIFY | SAVEPOINT |
| BOOLEAN | MOVE | SEARCH |
| BREADTH | MULTISET | SENSITIVE |
| CALL | NEW | SEQUENCE |
| CLASS | NEW_TABLE | SIGNAL |
| COMPLETION | NONE | SIMILAR |
| CONSTRUCTOR | OFF | SPECIFIC |
| CYCLE | OID | SQLEXCEPTION |
| DATA | OLD | SQLWARNING |
| DEPTH | OLD_TABLE | STRUCTURE |
| DEREF | OPERATION | TEMPLATE |
| DESTROY | OPERATORS | TEST |
| DESTRUCTOR | OTHERS | THAN |
| DICTIONARY | OUT | THERE |
| DO | PARAMETERS | TRIGGER |
| EACH | PENDANT | TUPLE |
| ELEMENT | PREORDER | TYPE |
| ELSEIF | PRIVATE | VARIABLE |
| EQUALS | PROTECTED | VARIANT |
| FUNCTION | RECURSIVE | VIRTUAL |
| GENERAL | REF | VISIBLE |
| IF | REFERENCING | WAIT |
| IGNORE | REPRESENTATION | WHILE |
| INOUT | RESIGNAL | WITHOUT |

| | |
|---|---|
| INSTEAD | RETURN |
| LEAVE | |

### F.4.4 VAX MACRO Reserved Symbols

The following VAX MACRO reserved symbols cannot be used as aliases for the SQL precompiler for SQL module processor (the ALIAS or DBHANDLE is used to name a PSECT or GLOBAL SYMBOL):

- IV
- R0, R1, R2 . . . R15
- AP
- SP
- FP

These VAX MACRO reserved symbols cannot be used in generated code. No warning message is generated because the language precompilers do not check for these reserved symbols.

## F.5 Punctuation Changes

The following changes apply to punctuation marks used in SQL.

### F.5.1 Single Quotation Marks Required for String Literals

Use single ( ′ ) instead of double ( " ) quotation marks to delimit a string literal. SQL flags literals enclosed within double quotation marks with an informational, compile-time, diagnostic message stating that this is nonstandard usage. This message will appear even when you have specified that SQL not notify you of syntax that is not ANSI/ISO SQL standard.

### F.5.2 Double Quotation Marks Required for ANSI/ISO SQL Delimited Identifiers

The leftmost name pair in a qualified name for a multischema object is a **delimited identifier**. You must enclose a delimited identifier within double quotation marks and use only uppercase characters. You must enable ANSI/ISO SQL quoting rules to use delimited identifiers. For more information, see Section 2.2.3.

### F.5.3 Colons Required Before Host Language Variables in SQL Module Language

In SQL module language statements, Oracle Rdb recommends that you precede parameters with a colon (:) to distinguish them from column or table names. These colons are currently optional in SQL, but are required by the ANSI/ISO SQL standard. SQL may require these colons in a future version of Oracle Rdb.

## F.6 Suppressing Diagnostic Messages

In interactive SQL, use the SET WARNING NODEPRECATE statement to suppress the diagnostic messages about deprecated features. For more information, see the SET Statement.

If you are using the SQL precompiler, you can suppress the diagnostic messages about deprecated features by using the SQLOPTIONS=WARN=(NODEPRECATE) qualifier in the precompiler command line. For details, see Section 4.3 and Section 4.4.

If you are using SQL module language, you can suppress the diagnostic messages about deprecated features by using the WARN=(NODEPRECATE) qualifier in the module language command line. For details, see Section 3.5 and Section 3.6.

# G
# Oracle7 SQL Functions

SQL functions have been added to the OpenVMS Oracle Rdb SQL interface for convergence with Oracle7 SQL. Complete descriptions of these functions can be found in the *Oracle7 Server SQL Language Reference Manual.* These functions are not available on the Digital UNIX platform.

Table G–1 describes the new functions that are built into the Oracle Rdb SQL interface:

**Table G–1 Built-In Oracle7 SQL Functions**

| Function Name | Description |
|---|---|
| CONCAT (s1,s2) | Returns s1 concatenated with s2.<br>CONCAT is functionally equivalent to the concatenation operator ( \| \| ) in Oracle Rdb. For example: |

```
SQL> SELECT DISTINCT
cont>    CONCAT (
cont>       CONCAT (
cont>          CONCAT (e.last_name, 'has a '),
cont>       d.degree),
cont>    ' degree')
cont> FROM employees e, degrees d
cont> WHERE e.employee_id = d.employee_id
cont> LIMIT TO 5 ROWS;

 Ames          has a MA  degree
 Ames          has a PhD degree
 Andriola      has a MA  degree
 Andriola      has a PhD degree
 Babbin        has a MA  degree
5 rows selected
```

(continued on next page)

**Table G–1 (Cont.) Built-In Oracle7 SQL Functions**

| Function Name | Description |
|---|---|
| CONVERT (str, dest_char_set) | Converts a character string to the specified character set.<br><br>You cannot specify the source character set as you can with Oracle7. dest_char_set must be a character set supported by Oracle Rdb.<br><br>For example: |

```
SQL> SELECT CONVERT (english, RDB$SHIFT_JIS)
cont> FROM colours;
 Black
 White
 Blue
 Red
 Yellow
 Green
6 rows selected
```

| Function Name | Description |
|---|---|
| DECODE (expr,srch1,res1, [,srch2,res2, . . . ,srchn,resn] [,default]) | Compares expr to srch1 through srchn until a match is found. When a match is found, DECODE returns the corresponding result in resn. If no match is found, DECODE returns the default if specified, null if not. For example: |

```
SQL> SELECT employee_id, last_name, first_name,
cont> DECODE (status_code, '1', 'Full time',
cont>                      '2', 'Part time')
cont> FROM employees
cont> LIMIT TO 5 ROWS;
 EMPLOYEE_ID   LAST_NAME       FIRST_NAME
 00165         Smith           Terry        Part time
 00190         O'Sullivan      Rick         Full time
 00187         Lasch           Stan         Full time
 00169         Gray            Susan        Full time
 00176         Hastings        Norman       Full time
5 rows selected
```

(continued on next page)

**Table G–1 (Cont.) Built-In Oracle7 SQL Functions**

| Function Name | Description |
|---|---|
| SYSDATE | Returns the current date and time. Requires no arguments.<br><br>SYSDATE is a synonym for CURRENT_TIMESTAMP. As with CURRENT_ TIMESTAMP, the return result of SYSDATE is affected by the setting of the SET DEFAULT DATE FORMAT statement as shown in the following example:<br><br>`SQL> SET DEFAULT DATE FORMAT 'SQL92'`<br>`SQL> SELECT SYSDATE, CURRENT_TIMESTAMP`<br>`cont>  FROM RDB$DATABASE;`<br><br>` 1995-08-21 15:21:05.29   1995-08-21 15:21:05.29`<br>`1 row selected`<br>`SQL> SET DEFAULT DATE FORMAT 'VMS'`<br>`SQL> SELECT SYSDATE, CURRENT_TIMESTAMP`<br>`cont>  FROM RDB$DATABASE;`<br><br>` 21-AUG-1995 15:21:24.83   21-AUG-1995 15:21:24.83`<br>`1 row selected` |

Optionally, you can install the functions listed in Table G–2 in your database from interactive SQL as shown in the following examples.

If you have installed the standard version of Oracle Rdb, use the following statement:

```
SQL> ATTACH 'FILENAME mydatabase';
SQL> @SYS$LIBRARY:SQL_FUNCTIONS.SQL
```

If you have installed the multiversion variant, the file is named SQL_FUNCTIONSnn.SQL, where “nn” is the version number. For example, use the following statement:

```
SQL> ATTACH 'FILENAME mydatabase';
SQL> @SYS$LIBRARY:SQL_FUNCTIONS70.SQL
```

If you wish to use a character set other than DEC_MCS with the installable functions, you must first define the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain as a character type using the desired character set before executing the preceding statements. Similarly, if you wish to use a date data type other than DATE VMS with the installable functions, you must first define the RDB$ORACLE_SQLFUNC_DATE_DOM domain as a date data type before executing the preceding statements.

Using the standard version, you use the following statements:

```
SQL> ATTACH 'FILENAME mydatabase';
SQL> CREATE DOMAIN RDB$ORACLE_SQLFUNC_VCHAR_DOM VARCHAR(2000)
cont> CHARACTER SET KANJI;
SQL> CREATE DOMAIN RDB$ORACLE_SQLFUNC_DATE_DOM DATE ANSI;
SQL> @SYS$LIBRARY:SQL_FUNCTIONS.SQL
```

If you choose, you may remove the installable functions from your database at a later time. However, you must release any dynamic SQL statements and disconnect any sessions that reference any of these functions before you can remove the functions. Use the following statements from interactive SQL if you wish to remove the installable functions from your database:

```
SQL> ATTACH 'FILENAME mydatabase';
SQL> @SYS$LIBRARY:SQL_FUNCTIONS_DROP.SQL
```

If you are using the multiversion variant, use the file SYS$LIBRARY:SQL_FUNCTIONS_DROPnn.SQL, where “nn” is the version number.

Table G–2 gives a brief description of each of the functions that you can optionally install in your database. For a complete description, refer to the *Oracle7 Server SQL Language Reference Manual*.

**Table G–2 Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| ABS (n) | Returns the absolute value of n. | |
| ADD_MONTHS (d,n) | Returns the date d plus n months. | d must be of the same date data type as the RDB$ORACLE_SQLFUNC_DATE_DOM domain, which is bound when you install the Oracle7 SQL functions. |
| ASCII (str) | Returns the decimal representation of the first character of its argument. | str must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. |
| CEIL (n) | Returns the smallest integer greater than or equal to n. | |

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| CHR (n) | Returns the character having the binary equivalent to n. | The returned value is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM, the character set of which is bound when you install the Oracle7 SQL functions. In addition, only 1 octet (byte) of data is encoded. |
| COS (n) | Returns the cosine of n (an angle expressed in radians). | |
| COSH (n) | Returns the hyperbolic cosine of n (an angle expressed in radians). | |
| EXP (n) | Returns e raised to the nth power (e=2.71828183 . . . ). | |
| FLOOR (n) | Returns the largest integer equal to or less than n. | |
| GREATEST (v1,v2) | Returns the greater of v1 or v2. | You must specify exactly 2 numeric arguments. Arguments of other types or more than 2 arguments are not supported at this time. |
| HEXTORAW (str) | Converts its argument containing hexadecimal digits to a raw character value. | str must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. |
| INITCAP (str) | Returns the string argument, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by non-alphanumeric characters. | str must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. |

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| INSTR (s1,s2,n,m) | Searches s1 beginning with its nth character and returns the character position of the mth occurrence of s2 or 0 if s2 does not occur m times. If n < 0, the search starts at the end of s1. | s1 and s2 must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. There are no defaults for n and m as with Oracle7.[1] |
| INSTRB (s1,s2,n,m) | Searches s1 beginning with its nth octet and returns the octet position of the mth occurrence of s2 or 0 if s2 does not occur m times. If n < 0, the search starts at the end of s1. | s1 and s2 must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. There are no defaults for n and m as with Oracle7.[1] |
| LAST_DAY (d) | Returns the last day of the month that contains d. | d must be of the same date data type as the RDB$ORACLE_SQLFUNC_DATE_DOM domain, which is bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_DATE_DOM. |
| LEAST (v1,v2) | Returns the lessor of v1 or v2. | You must specify exactly 2 numeric arguments. Arguments of other types or more than 2 arguments are not supported at this time. |
| LENGTH (str) | Returns the length of str in characters. | str must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. |

[1]If you have set your dialect to ORACLE LEVEL1 and have omitted a parameter for this function, the SQL compiler provides the default value for the parameter. See the *Oracle7 Server SQL Language Reference Manual* for information on the default values. Oracle Rdb recommends that you use the versions of these functions provided by Oracle Rdb.

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| LENGTHB (str) | Returns the length of str in bytes. | str must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. |
| LN (n) | Returns the natural logarithm of n where n is greater than 0. | |
| LOG (n,m) | Returns the logarithm base m of n. The base m can be any positive number other than 0 or 1 and n can be any positive number. | |
| LPAD (s,l,p) | Returns s left-padded to length l with the sequence of characters in p. If s is longer than l, this function returns that portion of s that fits in l. | s and p must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for p as with Oracle7.[1] |
| LTRIM (s1,s2) | Removes characters from the left of s1, with initial characters removed up to the first character not in s2. | s1 and s2 must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for s2 as with Oracle7.[1] |
| MOD (n,m) | Returns the remainder of m divided by n. Returns m if n is 0. | |

[1]If you have set your dialect to ORACLE LEVEL1 and have omitted a parameter for this function, the SQL compiler provides the default value for the parameter. See the *Oracle7 Server SQL Language Reference Manual* for information on the default values. Oracle Rdb recommends that you use the versions of these functions provided by Oracle Rdb.

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| MONTHS_ BETWEEN (d1,d2) | Returns the number of months between dates d1 and d2. | d1 and d2 must be of the same date data type as the RDB$ORACLE_SQLFUNC_ DATE_DOM domain, which is bound when you install the Oracle7 SQL functions. |
| NEW_TIME (d1,z1,z2) | Returns the date and time in time zone z2 when the date and time in time zone z1 is d. Time zones z1 and z2 can be: AST, ADT, BST, BDT, CST, CDT, EST, EDT, GMT, HST, HDT, MST, MDT, NST, PST, PDT, YST, or YDT. | d1 must be of the same date data type as the RDB$ORACLE_ SQLFUNC_DATE_DOM domain, which is bound when you install the Oracle7 SQL functions. z1 and z2 must be of the same character set as the RDB$ORACLE_SQLFUNC_ VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_ DATE_DOM. |
| NEXT_DAY (d,dayname) | Returns the date of the first weekday named by dayname that is later than the date d. | d must be of the same date data type as the RDB$ORACLE_ SQLFUNC_DATE_DOM domain, which is bound when you install the Oracle7 SQL functions. dayname must be of the same character set as the RDB$ORACLE_SQLFUNC_ VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_ DATE_DOM. |
| POWER (m,n) | Returns m raised to the nth power. The base m and the exponent n can be any number but if m is negative, n must be an integer. | |

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| RAWTOHEX (str) | Converts its raw argument to a character value containing its hexadecimal equivalent. | str must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. |
| REPLACE (s1,s2,s3) | Returns s1 with every occurrence of s2 replaced by s3. | s1, s2, and s3 must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for s3 as with Oracle7.[1] |
| ROUND (n,m) | Returns n rounded to m places right of the decimal point. m can be negative to round off digits left of the decimal point. m must be an integer. | There is no default for m as with Oracle7.[1] |
| RPAD (s,l,p) | Returns s left-padded to length l with the sequence of characters in p. If s is longer than l, this function returns that portion of s that fits in l. | s and p must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for p as with Oracle7.[1] |

[1]If you have set your dialect to ORACLE LEVEL1 and have omitted a parameter for this function, the SQL compiler provides the default value for the parameter. See the *Oracle7 Server SQL Language Reference Manual* for information on the default values. Oracle Rdb recommends that you use the versions of these functions provided by Oracle Rdb.

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| RTRIM (s1,s2) | Returns s2 with final characters after the last character not in s2. | s1 and s2 must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for s2 as with Oracle7.[1] |
| SIGN (n) | If n < 0, the function returns -1. If n = 0, the function returns 0. If n > 0, the function returns 1. | |
| SIN (n) | Returns the sine of n (an angle expressed in radians). | |
| SINH (n) | Returns the hyperbolic sine of n (an angle expressed in radians). | |
| SQRT (n) | Returns the square root of n. The value of n cannot be negative. SQRT returns a double precision result. | |
| SUBSTR (s,p,l) | Returns a portion of s, l characters long, beginning at character position p. If p is negative, SUBSTR counts backward from the end of s. | s must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for l as with Oracle7.[1] |

[1]If you have set your dialect to ORACLE LEVEL1 and have omitted a parameter for this function, the SQL compiler provides the default value for the parameter. See the *Oracle7 Server SQL Language Reference Manual* for information on the default values. Oracle Rdb recommends that you use the versions of these functions provided by Oracle Rdb.

(continued on next page)

**Table G–2 (Cont.) Optional Oracle7 SQL Functions**

| Function Name | Description | Restrictions |
|---|---|---|
| SUBSTRB (s,p,l) | Same as SUBSTR, except p and l are expressed in octets (bytes) rather than characters. | s must be of the same character set as the RDB$ORACLE_SQLFUNC_VCHAR_DOM domain, which is also bound when you install the Oracle7 SQL functions. The value returned is of type RDB$ORACLE_SQLFUNC_VCHAR_DOM. There is no default for l as with Oracle7.[1] |
| TAN (n) | Returns the tangent of n (an angle expressed in radians). | |
| TANH (n) | Returns the hyperbolic tangent of n (an angle expressed in radians). | |
| TRUNC (n,m) | Returns n truncated to m decimal places. m can be negative to truncate (make zero) m digits to the left of the decimal point. | There is no default for m as with Oracle7.[1] |

[1] If you have set your dialect to ORACLE LEVEL1 and have omitted a parameter for this function, the SQL compiler provides the default value for the parameter. See the *Oracle7 Server SQL Language Reference Manual* for information on the default values. Oracle Rdb recommends that you use the versions of these functions provided by Oracle Rdb.

# Index