Oracle Rdb™

SQL Reference Manual Volume 4

Release 7.4.1.1 for HPE OpenVMS Industry Standard 64 for Integrity Servers and OpenVMS Alpha operating systems

September 2021

ORACLE®

SQL Reference Manual, Volume 4

Oracle Rdb Release 7.4.1.1 for HPE OpenVMS Industry Standard 64 for Integrity Servers and OpenVMS Alpha operating systems

Primary Author: Rdb Engineering and Documentation group

Contents

Index

Examples

Send Us Your Comments

Oracle Rdb for OpenVMS
Oracle SQL Reference Manual, Release 7.4.1.1

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.

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

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

Please direct all comments, and corrections to this email address: **infordb_us@oracle.com**.

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

Preface

This manual describes the syntax and semantics of 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 Rdb Installation and Configuration Guide*.

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

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

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Structure

This manual is divided into five volumes. Volume 1 contains Chapter 1 through Chapter 5 and an index. Volume 2 contains Chapter 6 and an index. Volume 3 containsChapter 7 and an index. Volume 4 contains Chapter 8 and an index. Volume 5 contains 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, 3, 4, and 5 of the *Oracle Rdb SQL Reference Manual*:

Chapter 1	Introduces SQL (structured query language) and briefly describes SQL functions. This chapter also describes conformance to the SQL database 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 Chapter 7 Chapter 8	Describe 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 standards to which Oracle Rdb conforms.
Appendix C	Describes the SQL Communications Area, the message vector, and 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 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 compatibility with Oracle Database SQL. This appendix also describes the SQL syntax for performing an outer join between tables using Oracle Database syntax.
Appendix H	Describes the Oracle Rdb system tables.
Appendix I	Describes information tables that can be used with Oracle Rdb.
Appendix J	Describes the Database vault feature.

Related Manuals

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

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

Conventions

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.

The following conventions are also used in this manual:

. . .	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.

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. Version 7.4 of Oracle Rdb software is often referred to as V7.4.
- 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 I64 refers to HPE OpenVMS Industry Standard 64 for Integrity Servers.
- OpenVMS means the OpenVMS I64 and OpenVMS Alpha operating systems.

8 SQL Statements

This chapter describes the syntax and semantics of 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.

Chapter 6 in Volume 2 describes the statements from ACCEPT to CREATE SCHEMA. Chapter 7 in Volume 3 describes the statements from CREATE SEQUENCE to GRANT.

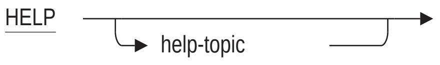
HELP Statement

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.
  - The responses to these prompts need only include sufficient unique characters to match the desired topic. For example the HELP CREATE SEQUENCE command will report this list of subtopics.

    ```
      Additional information available:

        Environment          Format     Arguments  Examples

    CREATE SEQUENCE Subtopic?
    ```

    In this case “En”, “F”, “A” and “Ex” are sufficient to uniquely identify a subtopic. Entering “*” will select and display all subtopics.
- 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:

  - Environment: This topic describes which SQL environment supports the statement.
  - Format: One or more syntax diagrams which detail the syntax choices.
  - Arguments: Subtopics describing the arguments.
  - Examples: One or more examples of the statement in use.

- SQL uses the default size of your terminal session; usually established by the DCL SET TERMINAL command. Many of the help topics contain more than will fix on a page and SQL will pause with the “Press RETURN for more:” prompt. Use the SQL SET PAGE LENGTH statement to adjust the page size for comfortable reading or set the page length to 32000 effectively disable this type of paging.

Example

Example 1: Obtaining online Help in SQL

```
SQL> HELP SELECT

SELECT

  Additional information available:

  General_Form         Singleton_Select

SELECT Subtopic?
```

IF Control Statement

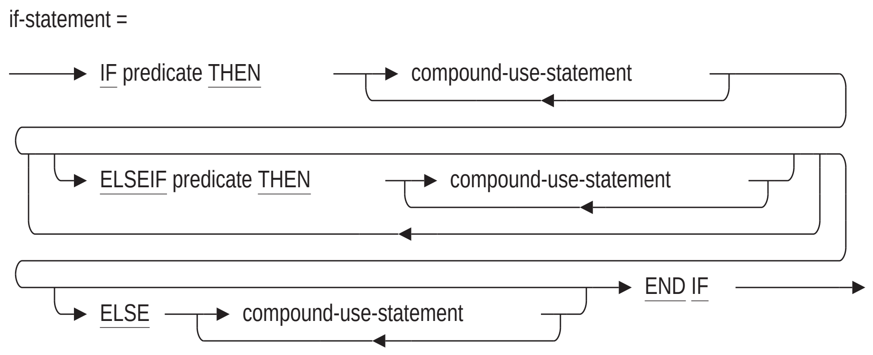
Executes one or more SQL statements conditionally. It then continues processing by executing any SQL statement that immediately follows the block.

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

compound-use-statement
See the Compound Statement for a description of the SQL statements that are valid in a compound statement.

END IF
Marks the end of an IF statement. Every IF statement must end with the END IF 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.

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 or ELSE clauses.

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.

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 branch 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.

If COMPRESSION was used during the EXPORT, the metadata in the interchange file defines the compression algorithm to be used by the IMPORT DATABASE statement and indicates which tables were compressed by the EXPORT DATABASE statement.

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 may not be allowed by an ALTER DATABASE statement. The IMPORT statement lets you:

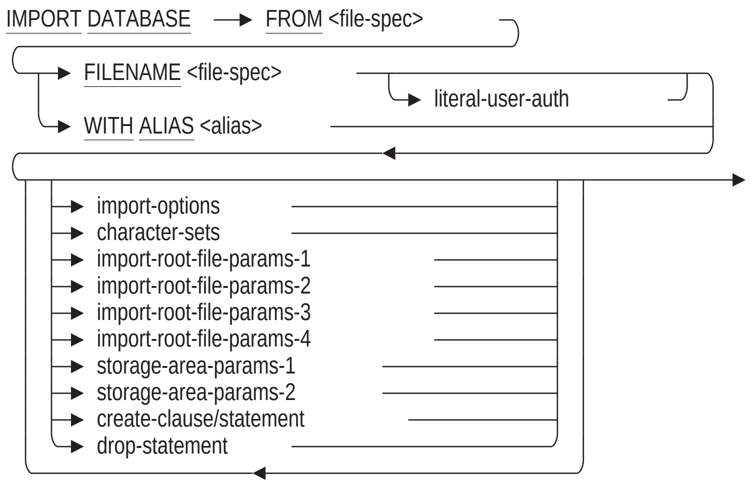
- Convert from a single-file to a multifile database, and vice versa.
- Change database root file parameters that you cannot change with the ALTER DATABASE statement:
  - COLLATING SEQUENCE
  - SEGMENTED STRING STORAGE AREA
  - PROTECTION IS ANSI/ACLS
  - DEFAULT STORAGE AREA
- Change storage area parameters that you cannot change with the ALTER DATABASE statement:
  - PAGE SIZE
  - PAGE FORMAT
  - THRESHOLDS
  - INTERVAL
  - FILENAME, 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.

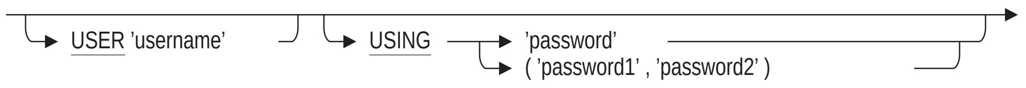
Environment

You can use the IMPORT statement in interactive SQL only.

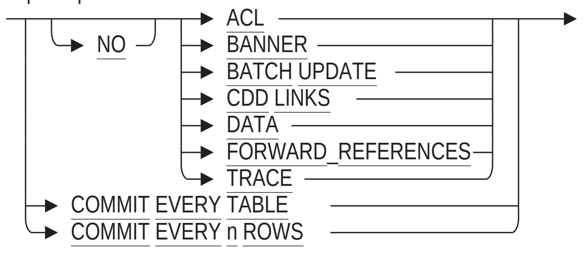
Format



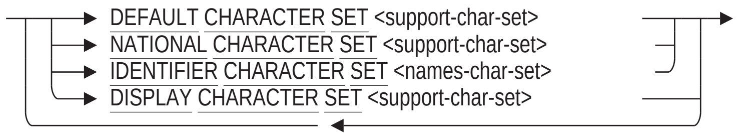
literal-user-auth =



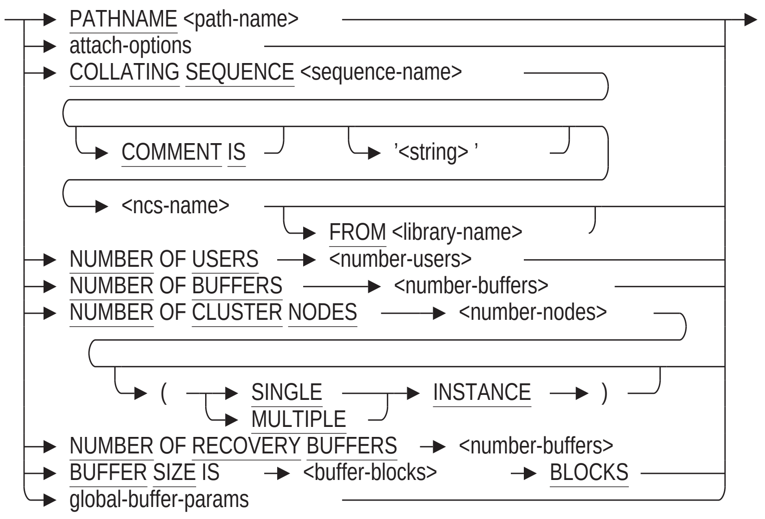
import-options=



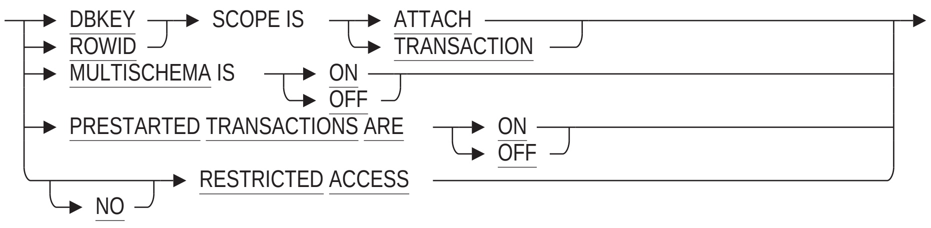
character-sets =


DEFAULT CHARACTER SET <support-char-set>
NATIONAL CHARACTER SET <support-char-set>
IDENTIFIER CHARACTER SET <names-char-set>
DISPLAY CHARACTER SET <support-char-set>

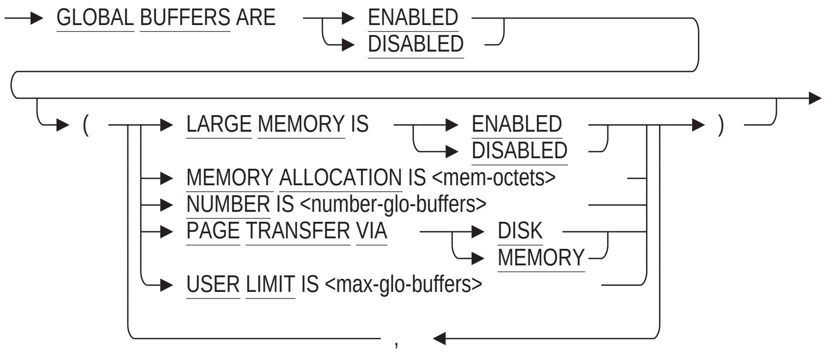
import-root-file-params-1 =



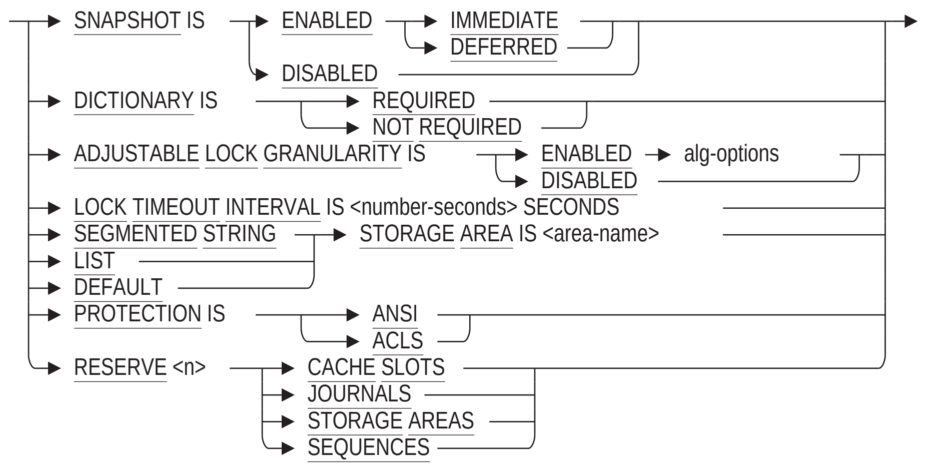
attach-options =



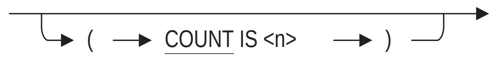
global-buffer-params=


GLOBAL BUFFERS ARE
ENABLED
DISABLED
(
LARGE MEMORY IS
ENABLED
DISABLED
)
MEMORY ALLOCATION IS <mem-octets>
NUMBER IS <number-glo-buffers>
PAGE TRANSFER VIA
DISK
MEMORY
USER LIMIT IS <max-glo-buffers>
,

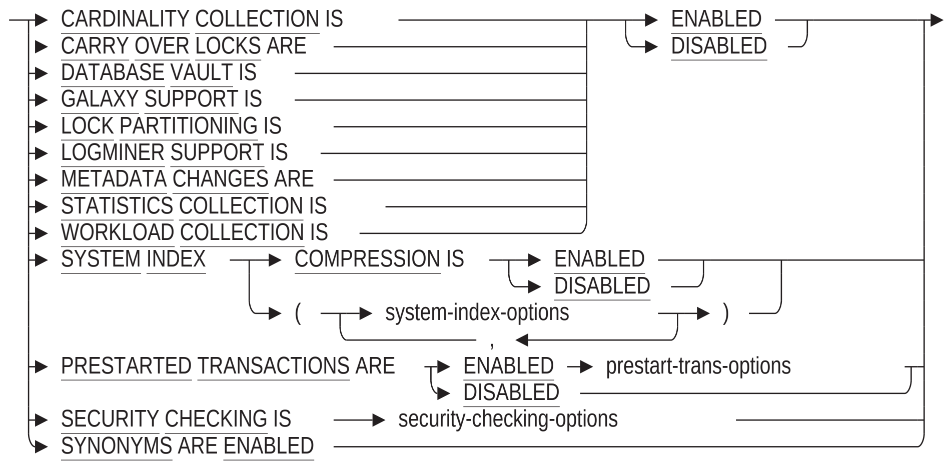
import-root-file-params-2 =



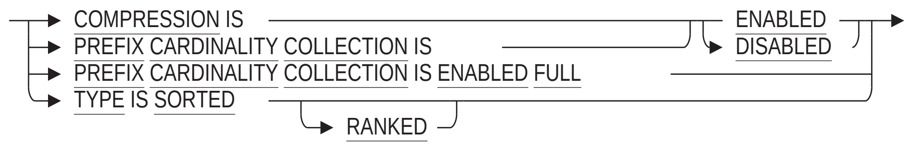
alg-options =



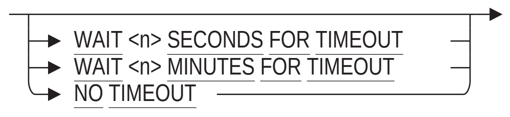
import-root-file-params-3 =



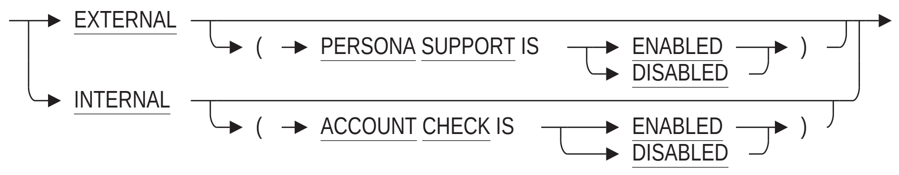
system-index-options =


COMPRESSION IS
PREFIX CARDINALITY COLLECTION IS
ENABLED
DISABLED
PREFIX CARDINALITY COLLECTION IS ENABLED FULL
TYPE IS SORTED
RANKED

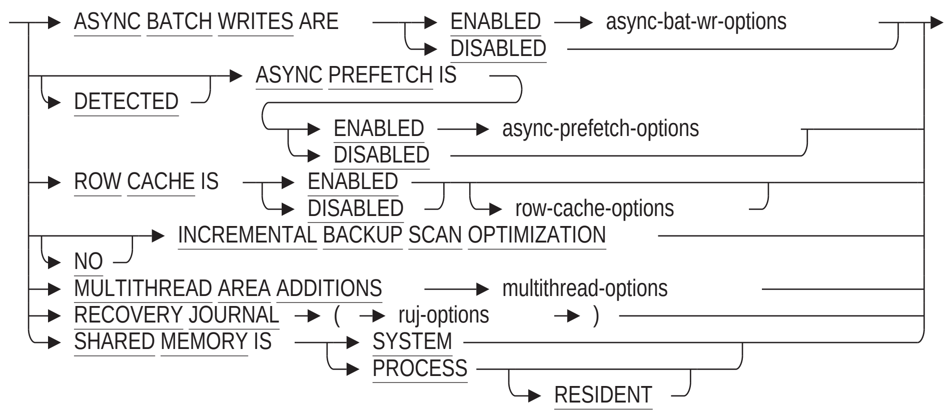
prestart-trans-options =



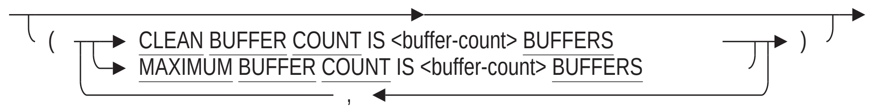
security-checking-options =


EXTERNAL
(PERSONA SUPPORT IS ENABLED DISABLED)
INTERNAL
(ACCOUNT CHECK IS ENABLED DISABLED)

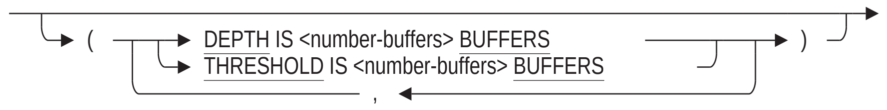
import-root-file-params-4 =


ASYNC BATCH WRITES ARE ENABLED async-bat-wr-options
DISABLED
ASYNC PREFETCH IS
DETECTED
ENABLED async-prefetch-options
DISABLED
ROW CACHE IS ENABLED
DISABLED row-cache-options
INCREMENTAL BACKUP SCAN OPTIMIZATION
NO
MULTITHREAD AREA ADDITIONS multithread-options
RECOVERY JOURNAL (ruj-options)
SHARED MEMORY IS SYSTEM
PROCESS
RESIDENT

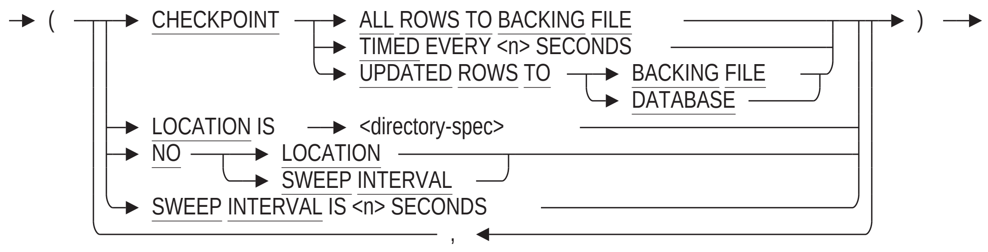
asynch-bat-wr-options =


(CLEAN BUFFER COUNT IS <buffer-count> BUFFERS)
MAXIMUM BUFFER COUNT IS <buffer-count> BUFFERS
,

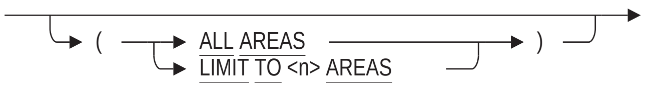
async-prefetch-options =


(DEPTH IS <number-buffers> BUFFERS)
THRESHOLD IS <number-buffers> BUFFERS
,

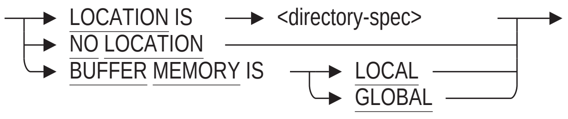
row-cache-options =


(CHECKPOINT ALL ROWS TO BACKING FILE)
TIMED EVERY <n> SECONDS
UPDATED ROWS TO BACKING FILE
DATABASE
LOCATION IS <directory-spec>
NO LOCATION
SWEEP INTERVAL
SWEEP INTERVAL IS <n> SECONDS
,

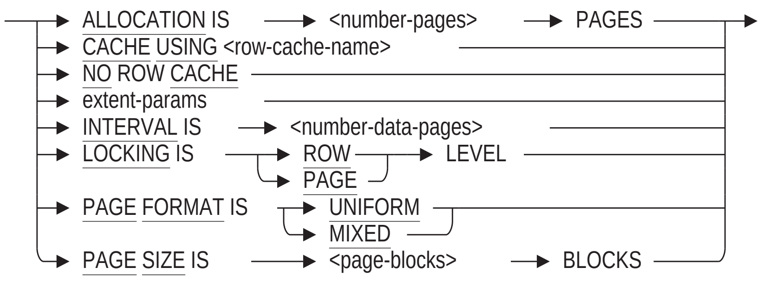
multithread-options =



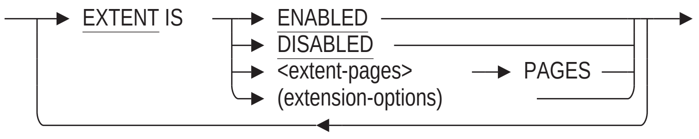
ruj-options =



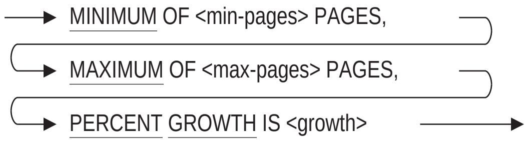
storage-area-params-1 =



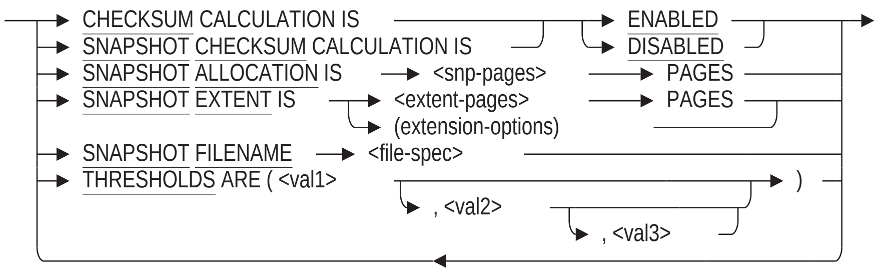
extent-params =



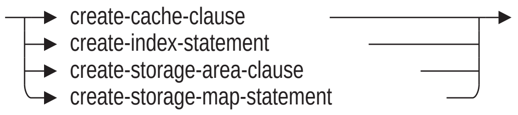
extension-options =


MINIMUM OF <min-pages> PAGES,
MAXIMUM OF <max-pages> PAGES,
PERCENT GROWTH IS <growth>

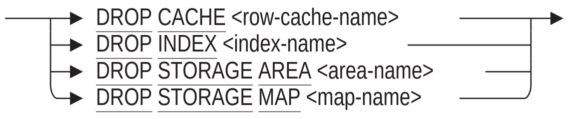
storage-area-params-2 =



create-clause/statement =



drop-statement =


DROP CACHE <row-cache-name>
DROP INDEX <index-name>
DROP STORAGE AREA <area-name>
DROP STORAGE MAP <map-name>

Arguments

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. The NO ACL option makes you the owner of the new database and creates default access control lists.

BANNER
NO BANNER
This clause requests that IMPORT display informational messages during the import of the database header, such as product identification, and values for some database parameters. The default is NO BANNER which will mean most IMPORT statements generate no output.

BATCH UPDATE
NO BATCH UPDATE
Specifies whether the IMPORT statement stores user data and indexes using batch-update transactions (BATCH UPDATE) or 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
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.

COMMIT EVERY TABLE
COMMIT EVERY n ROWS
Specifies whether the IMPORT statement commits entire tables, or commits a certain number of rows at regular intervals. If you use the COMMIT EVERY n ROWS clause, you can supply a value from 1 to 2147483647 for n.

The default is COMMIT EVERY TABLE. If you use the COMMIT EVERY n ROWS clause, the table will be left with a partial set of rows if the IMPORT process fails.

Note

If the table being imported includes a storage map with the PLACEMENT VIA INDEX clause, then the COMMIT EVERY clause is ignored for that table. A message is displayed to inform the database administrator of the tables that did not have COMMIT EVERY applied. This condition is shown in Example 6.

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.

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

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.

DROP CACHE row-cache-name
Prevents the specified row area from being imported.

DROP INDEX index-name
Prevents the specified index from being imported.

DROP STORAGE AREA area-name
Prevents the specified storage area from being imported.

DROP STORAGE MAP map-name
Prevents the specified storage map from being imported.

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
- Directory: the current directory for the process
- File name: the alias (if you omit the FILENAME argument, you must specify the WITH ALIAS clause)

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.

FORWARD_REFERENCES
NOFORWARD_REFERENCES
The EXPORT interchange file contains declarations of all routines that will be referenced by other definitions. The default is to declare the interfaces to those

routines prior to creating domains, tables, views, triggers, functions, procedures and modules that may need them. The default is FORWARD_REFERENCES.

Use NO FORWARD_REFERENCES to disable these declarations. However, this may result in definition failures during the IMPORT.

If you include the FORWARD_REFERENCES option on the IMPORT command line then informational messages will be generated for each declared routine.

FROM file-spec
Names the interchange .rbr file that the IMPORT statement uses as a source to create a new database.

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.

For more information on 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.

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

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

order-by-clause
See Section 2.8.1 for information about the ORDER BY clause.

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.

select-clause
See Section 2.8.1 for information about the SELECT clause.

storage-area-params1
storage-area-params2
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.

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 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*.

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

USING ’password’
USING (’password’, ’password’)
A character string literal that specifies the user’s password for the user name specified in the USER clause. If the user requires two passwords, then specify both values in parenthesis and separated by a comma.

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.

Usage Notes

- You must be granted the special rights identifier RDBVMS$DATABASE_VAULT_MANAGER to execute the DATABASE VAULT clause. Reference Appendix J, Guide to Database Management: Database Vault for more details.
- IMPORT executes two phases when importing a database:
  1. Create the database using the definitions saved in the interchange (.rbr) file, unless they were replaced or dropped by the IMPORT statement.
  2. Create all the metadata:
     - The database access control and security information
     - All synonyms used in the database
     - All roles, users, and profiles
     - All catalog information for a multischema database
     - All schema information for a multischema database
     - The LIST STORAGE MAP
     - All sequences
     - All collating sequences
     - All forward references to routines
     - All domain definitions
     - All external routines
     - All tables

       For each table the following actions are performed:
       * If a PLACEMENT VIA INDEX is defined, it will be created
       * Create the storage map
       * Import data for the current table if required
       * Create all indexes for the current table
     - All view definitions
     - All constraint definitions
     - All trigger definitions
     - All stored modules
     - All outlines

     The import process commits frequently to preserve any successfully executed definitions. A commit is performed after table load and each index creation to limit the size of the recovery unit journal (.ruj). Define the logical name RDMS$SET_FLAGS to the value "TRANSACTION" to see the transaction activity during the import process.

- If you wish to restructure an existing database with the EXPORT and IMPORT statements and keep database system files in the same directory, the Oracle recommends the following sequence:

  1. RMU Backup

     Preserve a copy of the original database in case of failure of the IMPORT command.

  2. EXPORT

     Save the database metadata and table data for subsequent IMPORT. Make sure that sufficient space exists for this export (rbr) file.

  3. DROP DATABASE

     If you do not delete the database, the IMPORT statement fails because the database storage areas files already exist.

  4. IMPORT

     Using the saved file rebuild the database, adding any changed database parameters, storage areas and index definitions in the IMPORT command. Note that after image journal file from the original database can not be applied to this totally new database.

  5. 5. RMU Backup the new database

     Preserve a copy of the new database for use with RMU/RESTORE and RMU/RECOVER.

- When importing the CDD$COMPATIBILITY repository, use the DICTIONARY IS NOT USED clause to prevent SQL from attempting to use the repository.
- The CREATE STORAGE AREA, CREATE STORAGE MAP, and CREATE INDEX statements within an IMPORT statement can refer to storage areas, storage maps, and indexes that existed in the original database. When they refer to existing elements, the IMPORT statement replaces those elements of the same name using the characteristics specified in the CREATE statements (or the database system defaults for characteristics not specified in the CREATE statements).
- The IMPORT statement creates a new database that inherits the characteristics of the database that was the source for the .rbr file used by the IMPORT statement. Only the elements you create will differ from the original database.
- If you do not specify a page size when creating a storage area with the IMPORT statement, the page size is inherited from RDB$SYSTEM.
- 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.

     You can use the command RMU Dump Export command with the Nodata qualifier to extract the metadata in the import interchange file to see the names of the storage areas in the database.
  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.
- 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.
- See the *Oracle Rdb Guide to Database Maintenance* for a complete discussion of when to use the IMPORT, EXPORT, and ALTER DATABASE statements.
- 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.
- If 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.
- 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.

- A node specification may only be specified for the root FILENAME clause of the IMPORT DATABASE statement.

  This means that the directory or file specification specified with the following clauses can only be a device, directory, file name, and file type:

  - LOCATION clause of the ROW CACHE IS ENABLED, RECOVERY JOURNAL, ADD CACHE, and CREATE CACHE clauses
  - SNAPSHOT FILENAME clause
  - FILENAME and SNAPSHOT FILENAME clauses of the CREATE STORAGE AREA clause

- If the interchange file is being used by a previous version of Oracle Rdb, the NOFORWARD_REFERENCES clause should be used on EXPORT to prevent the dependency information being exported. In addition, the dependency information in the interchange file can be ignored by Oracle Rdb Release 7.1.0.4 and later versions using the NOFORWARD_REFERENCES clause of the IMPORT DATABASE statement.

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> export database
cont>     filename PERSONNEL
cont>     into PERS;
SQL>
SQL> import database
cont>     from PERS
cont>     filename MF_PERSONNEL
cont>     default storage area MFP0
cont>     create storage area MFP0
cont>         filename MFP0_DEFAULT
cont>         page format is UNIFORM
cont>     create storage area MFP1
cont>         filename MFP1
cont>     create storage area MFP2
cont>         filename MFP2
cont>     create storage map EMPLOYEES_MAP
cont>         for EMPLOYEES
cont>         store randomly across (MFP1, MFP2);
SQL>
SQL> show storage area;
Storage Areas in database with filename MF_PERSONNEL
     MFP0                            Default storage area
     MFP1
     MFP2
     RDB$SYSTEM                      List storage area.
```

Note that the storage area RDB$SYSTEM was created implicitly in this example. The database administrator could add a CREATE STORAGE AREA RDB$SYSTEM clause to this IMPORT example so that the name, location and space allocation for the RDB$SYSTEM area can be controlled.

Example 2: Importing a database created with ANSI/ISO-style privileges

This example imports a database originally created using ACLS style protection to create a new database with ANSI style protections.

```
SQL> import database
cont>     from PERS
cont>     alias NEW_PERS
cont>     filename MF_PERSONNEL
cont>     protection is ANSI
cont> ;
SQL> show protection on database NEW_PERS;
Protection on Alias NEW_PERS
[DEV,SMITH]:
  With Grant Option:        SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,OPERATOR,DBADM,SECURITY,DISTRIBTRAN
  Without Grant Option:     NONE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
SQL>
SQL> show protection on table NEW_PERS.EMPLOYEES;
Protection on Table NEW_PERS.EMPLOYEES
[DEV,SMITH]:
  With Grant Option:        SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,REFERENCES
  Without Grant Option:     NONE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
```

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);
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>
```

Example 4: Reserving Sequence Slots During an Import Operation

```
SQL> IMPORT DATABASE FROM MF_PERSONNEL.RBR
cont> FILENAME 'mf_personnel.rdb' BANNER
cont> RESERVE 64 SEQUENCES;
   .
   .
   .
Unused Sequences were 32 now are 64
IMPORTing STORAGE AREA: RDB$SYSTEM
IMPORTing STORAGE AREA: DEPARTMENTS
IMPORTing STORAGE AREA: EMPIDS_LOW
```

Example 5: Specifying the BANNER option

```
SQL> import data from x file mf_personnel BANNER;
Exported by Oracle Rdb V7.2-501 Import/Export utility
A component of Oracle Rdb SQL V7.2-501
Previous name was mf_personnel
It was logically exported on 29-MAY-2003 12:32
Multischema mode is DISABLED
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 JOURNAL ALLOCATION is 512
Database JOURNAL EXTENSION is 512
Database BUFFER SIZE is 6 blocks
Database NUMBER OF BUFFERS is 20
Adjustable Lock Granularity is Enabled Count is 3
Database global buffering is DISABLED
Database number of global buffers is 250
Number of global buffers per user is 5
Database global buffer page transfer is via DISK
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
Statistics Collection is ENABLED
Unused Storage Areas are: 0
System Index Compression is DISABLED
Journal was Disabled
Unused Journals are: 1
Journal Backup Server was:   Manual
Journal Log Server was:      Manual
Journal Overwrite was:       Disabled
Journal shutdown minutes was 60
Asynchronous Prefetch is ENABLED
Async prefetch depth buffers is 5
Asynchronous Batch Write is ENABLED
Async batch write clean buffers is 5
Async batch write max buffers is 4
Lock Partitioning is DISABLED
Incremental Backup Scan Optim uses SPAM pages
```

```
Unused Cache Slots are: 1
Workload Collection is DISABLED
Cardinality Collection is ENABLED
Metadata Changes are ENABLED
Row Cache is DISABLED
Detected Asynchronous Prefetch is ENABLED
Detected Asynchronous Prefetch Depth Buffers is 4
Detected Asynchronous Prefetch Threshold Buffers is 4
Open is Automatic, Wait period is 0 minutes
Shared Memory is PROCESS
Unused Sequences are: 32
The Transaction Mode(s) Enabled are:
     ALL
IMPORTing STORAGE AREA: RDB$SYSTEM
IMPORTing STORAGE AREA: DEPARTMENTS
IMPORTing STORAGE AREA: EMPIDS_LOW
IMPORTing STORAGE AREA: EMPIDS_MID
IMPORTing STORAGE AREA: EMPIDS_OVER
IMPORTing STORAGE AREA: EMP_INFO
IMPORTing STORAGE AREA: JOBS
IMPORTing STORAGE AREA: MF_PERS_SEGSTR
IMPORTing STORAGE AREA: SALARY_HISTORY
IMPORTing table CANDIDATES
IMPORTing table COLLEGES
IMPORTing table DEGREES
IMPORTing table DEPARTMENTS
IMPORTing table EMPLOYEES
IMPORTing table JOBS
IMPORTing table JOB_HISTORY
IMPORTing table RESUMES
IMPORTing table SALARY_HISTORY
IMPORTing table WORK_STATUS
IMPORTing view CURRENT_SALARY
IMPORTing view CURRENT_JOB
IMPORTing view CURRENT_INFO
```

Example 6: Using the COMMIT EVERY option

```
SQL> import database
cont>     from 'TEST$DB_SOURCE:MF_PERSONNEL'
cont>     filename 'MF_PERSONNEL'
cont>
cont>     commit every 10 rows
cont>
cont>     create storage area DEPARTMENTS
cont>         filename 'DEPARTMENTS'
cont>         page format is mixed
cont>         snapshot filename 'DEPARTMENTS'
cont>     create storage area EMPIDS_LOW
cont>         filename 'EMPIDS_LOW'
cont>         page format is mixed
cont>         snapshot filename 'EMPIDS_LOW'
cont>     create storage area EMPIDS_MID
cont>         filename 'EMPIDS_MID'
cont>         page format is mixed
cont>         snapshot filename 'EMPIDS_MID'
cont>     create storage area EMPIDS_OVER
cont>         filename 'EMPIDS_OVER'
cont>         page format is mixed
cont>         snapshot filename 'EMPIDS_OVER'
   .
   .
   .
```

```
cont> ; ! end of import
Definition of STORAGE AREA RDB$SYSTEM overridden
Definition of STORAGE AREA MF_PERS_SEGSTR overridden
Definition of STORAGE AREA EMPIDS_LOW overridden
Definition of STORAGE AREA EMPIDS_MID overridden
Definition of STORAGE AREA EMPIDS_OVER overridden
Definition of STORAGE AREA DEPARTMENTS overridden
Definition of STORAGE AREA SALARY_HISTORY overridden
Definition of STORAGE AREA JOBS overridden
Definition of STORAGE AREA EMP_INFO overridden
COMMIT EVERY ignored for table EMPLOYEES due to PLACEMENT VIA INDEX processing
COMMIT EVERY ignored for table JOB_HISTORY due to PLACEMENT VIA INDEX processing
SQL>
```

INCLUDE Statement

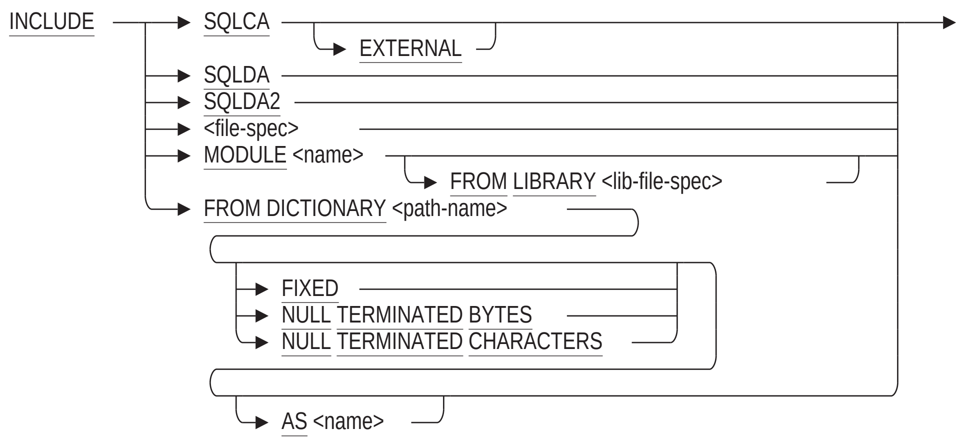
Inserts declarations or code into a precompiled host language program. You can use the INCLUDE statement to insert:

- 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
- Host language declarations for repository record definitions

Environment

You can use the INCLUDE statement 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

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.

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.

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.

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.

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.

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, including those repository objects 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.

MODULE FROM LIBRARY <library-file-spec>
This command will include the source text from the named text library. The text library should be created using the OpenVMS command LIBRARY/CREATE/TEXT. The name of the modules in that library can be determined using the LIBRARY/LIST/TEXT. It is posible that these modules are specifically named using the /MODULE qualifier on the LIBRARY command.

```
$ LIBRARY/CREATE/TEXT PERSONNEL_DEFS.TLB
$ LIBRARY/REPLACE/TEXT PERSONNEL_DEFS.TLB EMPS.LIB/MODULE=EMPLOYEES_REC
$ LIBRARY/REPLACE/TEXT PERSONNEL_DEFS.TLB SH.LIB/MODULE=SALARY_HISTORY_REC
$ LIBRARY/REPLACE/TEXT PERSONNEL_DEFS.TLB JH.LIB/MODULE=JOB_HISTORY_REC
$
```

To reference this text library, the application would use an INCLUDE statement as shown below:

```
EXEC SQL INCLUDE MODULE EMPLOYEES_REC FROM LIBRARY 'PERSONNEL_DEFS.TLB'
END-EXEC
```

MODULE <modulename>

This abbreviated statement defaults to using the text library named SQL$TEXT_LIBRARY in the default directory or referenced by the logical name SQL$TEXT_LIBRARY.

```
EXEC SQL INCLUDE MODULE EMPLOYEES_REC
END-EXEC
```

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.

NULL TERMINATED CHARACTERS

Specifies that CHAR fields from the dictionary are null-terminated, but the module processor interprets the length field as a character count. If *n* is the length in the dictionary, then the number of data bytes is *n*, and the length of the string is $n + 1$ bytes.

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 C.

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.

Usage Notes

- 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 Appendix C.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. 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.

- 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$RELATIONS that is subordinate to the database path name. When referencing these definitions the path name in the INCLUDE FROM DICTIONARY statement must include the RDB$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.
  ```
- Using the INCLUDE command makes any included text visible to the SQL Precompiler and also the target language. Use this command when you wish to make variable and record definitions visible to SQL or if the included text also contains EXEC SQL directives.
- The module included from a text library may not also include the INCLUDE file-spec statement nor the INCLUDE MODULE statement.
- The default file type for the text LIBRARY is .TLB
- If the text library is created with case sensitive names, then the MODULE name must be in quotes to preserve the case of the name.

  ```
  $ LIBRARY/CREATE=CASE_SENSITIVE:yes/TEXT PERSONNEL_DEFS.TLB
  $ LIBRARY/REPLACE/TEXT -
      PERSONNEL_DEFS.TLB -
      JH.LIB/MODULE="JobHistoryRecord"
  ```

  In such cases, the SQL$PRE command line, or the MODULE header must specify that QUOTING RULES are enabled to allow quoted names. This can be specified using /SQLOPTIONS qualifier to specify either ANSI_ IDENTIFIERS or ANSI_QUOTING, or compiling with a DECLARE MODULE statement in a context file.

  ```
  $       ! Use a context file and set SQL99 quoting rules
  $       CREATE CONTEXT_FILE.SQL
  declare module TESTING
      pragma (ident 'V1.00')
      quoting rules sql99;
  $       DEFINE/USER SQL$TEXT_LIBRARY INCLUDE_MODULE.TLB
  $       SQL$PRE/COBOL SAMPLE_APP CONTEXT_FILE.SQL
  ```

Examples

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 specifies 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
    FILENAME 'SQL$DATABASE'
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 PERS.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.
    STOP RUN.
ERR.
    DISPLAY "unexpected error ", sqlcode with conversion.
    CALL "SQL$SIGNAL".
```

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:

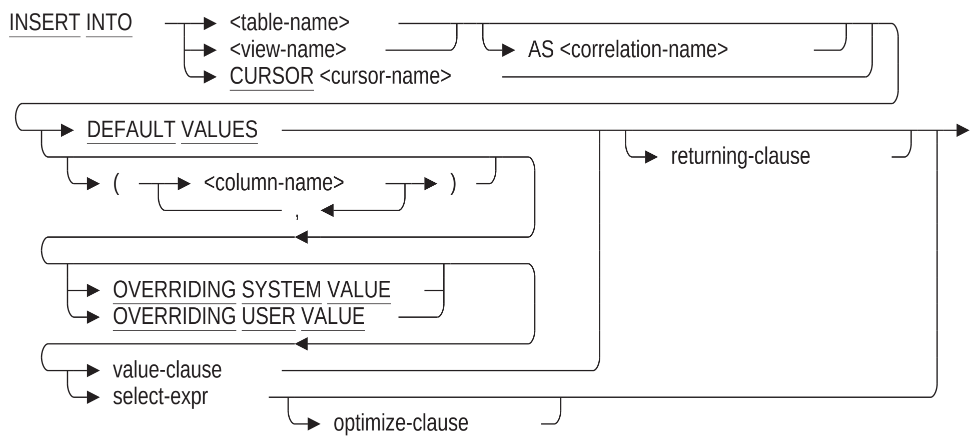
- The current transaction must not be read-only.
- You cannot specify a cursor name that refers to an update table cursor.
- Your cursor must specify an intermediate table.
- The value that you assign is appended to the end of the list.

Environment

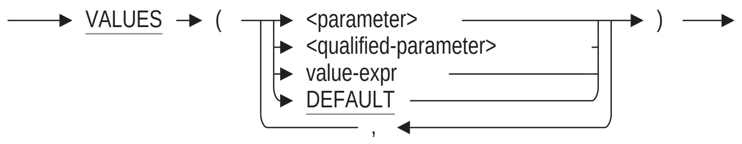
You can use the INSERT statement:

- 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

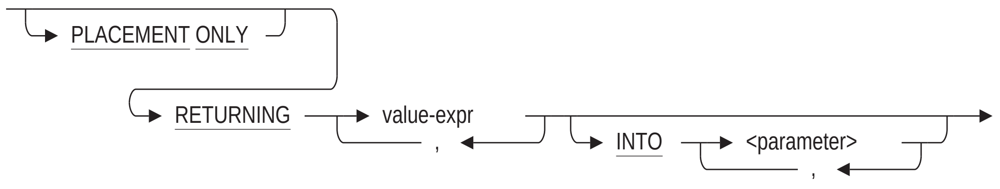
Format



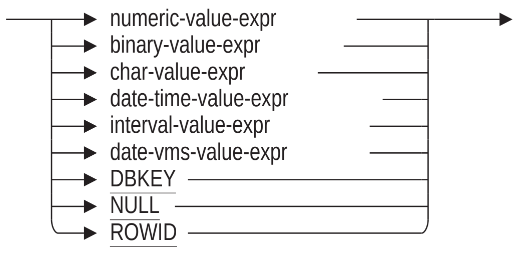
value-clause =



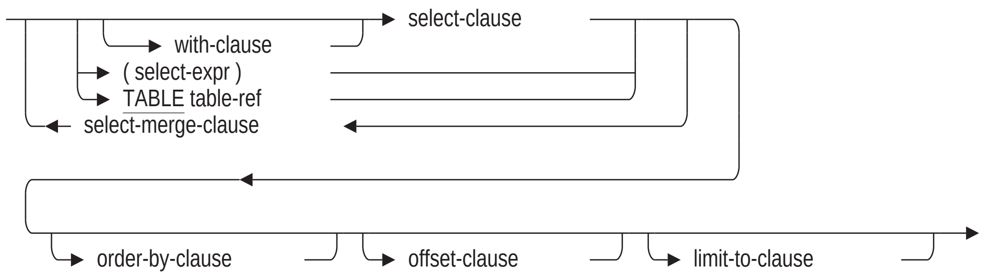
returning-clause =



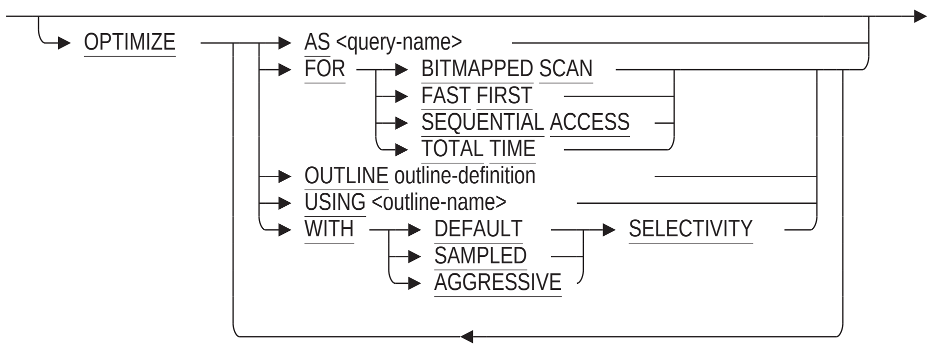
value-expr =



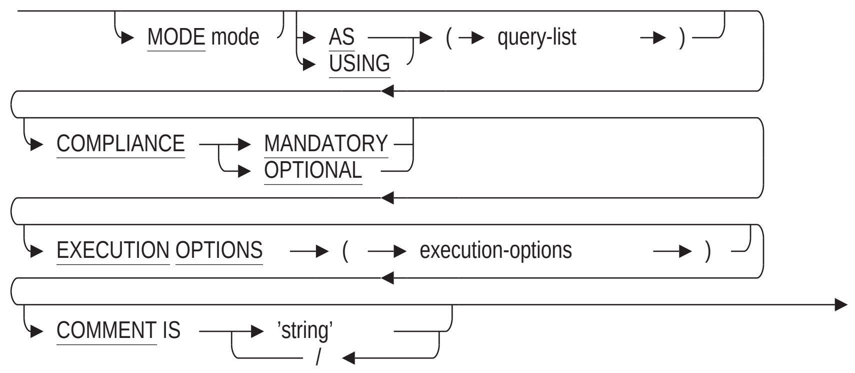
select-expr =



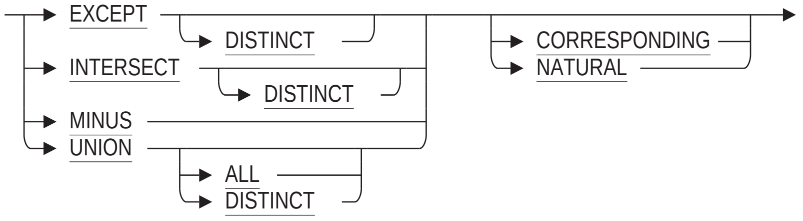
optimize-clause =



outline-definition =



select-merge-clause =



Arguments

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 as an AUTOMATIC INSERT AS column
- Defined with the IDENTITY attribute
- 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.

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.

DEFAULT
Forces the named column to assume the default value defined for that column (or NULL if none is defined).

If the DEFAULT clause is used in an INSERT statement then one of the following will be applied:

- If a DEFAULT attribute is present for the column then that value will be applied during INSERT.
- If an AUTOMATIC attribute is present for the column then that value will be applied during INSERT. This can only happen if the SET FLAGS 'AUTO_OVERRIDE' is used since during normal processing these columns are read-only.
- Otherwise a NULL will be applied during INSERT.

DEFAULT VALUES
Specifies that every column in the table be assigned a default value. Generated columns (such as AUTOMATIC INSERT AS, GENERATED, or IDENTITY) will be assigned the result of the value expression used to define the column. Otherwise NULL will be assigned for non-generated columns, or columns without a DEFAULT.

INTO parameter
Returns the values from the RETURNING clause into the specified list of parameters. There must exist one parameter for each value expression listed by the RETURNING clause.

This clause may be omitted for Interactive SQL which will then print the returned values.

INTO table-name
INTO view-name
The name of the target cursor, table or view to which you want to add a row.

limit-to-clause
See Section 2.8.1 for a description of the LIMIT TO expression.

OPTIMIZE AS query-name
The OPTIMIZE AS clause assigns a name to the query. Use the SET FLAGS 'STRATEGY' to see this name displayed.

OPTIMIZE FOR
The OPTIMIZE FOR clause specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- BITMAPPED SCAN

  Requests the Rdb query optimizer attempt to use BITMAPPED SCAN if there exists multiple supporting indices. This option is not compatible with SEQUENTIAL ACCESS.
- 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 canceled 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.

Choose either FAST FIRST or TOTAL TIME. If optimization strategy is not explicitly set, FAST FIRST is the default.

- SEQUENTIAL ACCESS

  Forces the use of sequential access. This is particularly valuable for tables that use the strict partitioning functionality. This option is not compatible with BITMAPPED SCAN.

- 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.

  Choose either FAST FIRST or TOTAL TIME. Queries inside compound statements default to TOTAL TIME.

OPTIMIZE OUTLINE outline-definition

The OPTIMIZE OUTLINE clause declares a temporary query outline to be used with the select expression.

See the CREATE OUTLINE Statement for more information on defining an outline.

See the CREATE OUTLINE Statement for more information on defining an outline.

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 WITH

Selects one of three optimization controls: DEFAULT (as used by previous versions of Rdb), AGGRESSIVE (assumes smaller numbers of rows will be selected), and SAMPLED (which uses literals in the query to perform preliminary estimation on indices).

The following example shows how to use this clause.

```
SQL> select * from employees where employee_id  > '00200'
cont>   optimize with sampled selectivity;
```

order-by-clause

See Section 2.8.1 for a description of the ORDER BY expression.

OVERRIDING

The OVERRIDING SYSTEM VALUE clause instructs Rdb that the GENERATED, IDENTITY or AUTOMATIC AS columns will be inserted with user supplied values and therefore no generated values will be created. Such a clause would be used if a table was being reloaded after maintenance and the database administrator wanted to retain the saved generated values.

The OVERRIDING USER VALUE clause instructs Rdb that the GENERATED, IDENTITY or AUTOMATIC AS columns will be generated by the database system and that any user supplied values will be ignored. Such a clause would be used when column names were wild carded by a SELECT clause and therefore avoids enumerating all non-generated column names.

Without this clause INSERT into AUTOMATIC, GENERATED (always) and IDENTITY (always) will result in a read-only column error.

PLACEMENT ONLY
This clause modifies the INSERT statement so that Oracle Rdb computes the approximate location, that is placement, of the inserted data. No actual update of the database occurs. Usually this clause is coupled with the RETURNING DBKEY OR RETURNING ROWID clause so that the target storage area, possibly the target page number can be collected by an application.

Use of this clause can improve bulk data loads by allowing applications to order the rows to be inserted by storage area number and page number prior to starting the bulk load. In this way the load program makes better use of the buffered pages by accessing them in a well defined order. For more information, see the *Oracle Rdb Guide to Database Design and Definition*.

The keyword ROWID is a synonym to the DBKEY keyword.

RETURNING value-expr
Returns the values of columns from the target table. If DBKEY is specified, this argument returns the database key (dbkey) of the row being added. When the DBKEY value is valid, subsequent queries can use the DBKEY value to access the row directly. The ROWID keyword is a synonym for the DBKEY keyword.

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.

select-clause
See Section 2.8.1 for a description of the SELECT expression.

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.

INSERT is one of the few SQL statements that allows you to specify a second database.

The number of columns in the result table must correspond to the number of updatable 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 updatable 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.

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.

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 updatable 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 updatable 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.

Values for IDENTITY, COMPUTED BY, and AUTOMATIC columns are not able to be inserted so these column types are not considered for the default column list.

See the SQL Online Help topic INSERT EXAMPLES for an example that shows an INSERT statement with a column select expression.

Usage Notes

- You must have the INSERT privilege on the target table or view in order to insert rows.
- The INSERT will execute BEFORE and AFTER INSERT triggers if they exist.
- 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. 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 store rows into
SQL> -- it will fail because no value is provided for EMPLOYEE_ID:
SQL> INSERT INTO TEMP (SUPERVISOR_ID) VALUES ('99999');
%RDB-E-INTEG_FAIL, violation of constraint JH_EMP_ID_EXISTS caused operation to fail
-RDB-F-ON_DB, on database RDB$DEFAULT_CONNECTION
SQL> ROLLBACK;
SQL>
```

- 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.

  INSERT is one of the few SQL statements that allows you to specify a second database. 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.

- Oracle Rdb uses the outline specified in the OPTIMIZE USING clause (if it exists) or as specified by the OPTIMIZE OUTLINE 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.

- If the target table of the INSERT statement has an IDENTITY column then a new value will be generated for the new row.

  In addition the CURRVAL pseudo column can be used with the name of the table to reference the new sequence number. For instance, this example assumes the table ORDER has a column defined with the IDENTITY attribute.

  ```
  SQL> insert into ORDER values (...);
  SQL> insert into ORDER_LINES (ORDER.CURRVAL, ...);
  SQL> insert into ORDER_LINES (ORDER.CURRVAL, ...);
  ```

  This example shows that the FOREIGN KEY value is selected using a reference to the table name followed by the CURRVAL pseudo column.

  However, the NEXTVAL pseudo column can not be used to fetch a new identity value. Only an INSERT on the table can generate a new identity value.

- If the INSERT on the table is rolled back or fails due to a constraint or trigger error condition, then the used identity values are discarded. If a row is deleted from the table, the identity value is not reused. For an exception to the reuse rule, see the usage note on TRUNCATE TABLE statement.

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>
```

Example 7: Inserting the user name and an amount into table columns:

```
SQL> CREATE TABLE TABLE1
cont> (ID CHAR(15),
cont>  AMOUNT INT(4));
SQL> INSERT INTO TABLE1 (ID, AMOUNT)
cont>           VALUES (USER, 1000);
SQL> SELECT * FROM TABLE1;
 ID                         AMOUNT
 ELLINGSWORTH            1000.0000
1 row selected
```

Example 8: Inserting a name and a column select expression into the same table columns used in the previous example:

```
SQL> INSERT INTO TABLE1 (ID, AMOUNT)
cont>            VALUES ('BROWN',
cont>                   (SELECT COUNT (*) FROM TABLE1));
SQL> SELECT * FROM TABLE1;
 ID                          AMOUNT
 HALVORSON                1000.0000
 BROWN                       1.0000
2 rows selected
```

Example 9: Inserting Default Values for Selected Columns

```
SQL> INSERT INTO DEPARTMENTS
cont> (DEPARTMENT_CODE, DEPARTMENT_NAME, BUDGET_ACTUAL)
cont> VALUES
cont> ('RECR','Recreation', DEFAULT);
1 row inserted
SQL> SELECT * FROM DEPARTMENTS WHERE DEPARTMENT_CODE='RECR';
 DEPARTMENT_CODE   DEPARTMENT_NAME                  MANAGER_ID
   BUDGET_PROJECTED   BUDGET_ACTUAL
 RECR              Recreation                       NULL
               NULL            NULL
1 row selected
```

Example 10: Inserting a Row of All Default Values into a Table

```
SQL> INSERT INTO CANDIDATES
cont> DEFAULT VALUES;
1 row inserted
SQL> SELECT * FROM CANDIDATES
cont> WHERE LAST_NAME IS NULL;
 LAST_NAME        FIRST_NAME   MIDDLE_INITIAL
   CANDIDATE_STATUS
      RESUME
 NULL             NULL         NULL
   NULL
   >>
   >>
   >>
      NULL
1 row selected
```

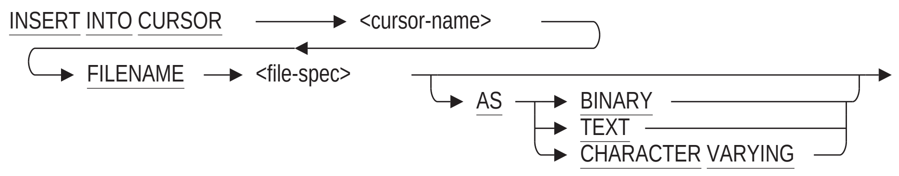
INSERT from FILENAME Statement

Loads a column of the LIST OF BYTE VARYING data type from a text or binary file without needing to use special application code. The specified file is opened and each row is read and stored in the LIST OF BYTE VARYING column specified by the list cursor.

Environment

You can use the INSERT statement in interactive SQL only.

Format



Arguments

AS BINARY
AS CHARACTER VARYING
AS TEXT
Specifies whether the file specified with the FILENAME clause contains these types of data:

- BINARY

  Used to load unformatted data such as images and audio files. The contents are broken into 512 octet segments during INSERT.

- CHARACTER VARYING

  Used to load text but with no terminator. The contents are written one line to a segment.

- TEXT

  Used to load text, a terminator is added to each segment loaded. The contents are written one line to a segment with trailing terminators carriage return (CR) and line feed (LF).

FILENAME filespec
The specification for the file that you want to load into the LIST OF BYTE VARYING column.

INSERT INTO CURSOR cursor-name
The name of the target list cursor to which you want to add a list segment.

Usage Notes

- When you use an INSERT from FILENAME statement to assign values to list segments:
  - The current transaction must be read/write.

  - Your cursor must specify an insert-only list cursor.

- Interactive SQL also reports the number of segments inserted, and the length of the longest segment. To disable this output use the SET DISPLAY NO ROW COUNT statement.
- The TEXT and CHARACTER VARYING source can contain segments of up to 65500 bytes in length. In prior releases the upper limit was 512 octets.

Example

Example 1: Adding a New Row Using Data from a Text File

```
SQL> -- Declare a table cursor.
SQL> DECLARE TABLE_CURSOR
cont>     INSERT ONLY TABLE CURSOR
cont>     FOR SELECT * FROM RESUMES;
SQL> -- Open table cursor and insert values.
SQL> OPEN TABLE_CURSOR;
SQL> INSERT INTO CURSOR TABLE_CURSOR
cont>     VALUES ('10065', NULL);
1 row inserted
SQL> -- Declare a list cursor.
SQL> DECLARE LIST_CURSOR
cont>     INSERT ONLY LIST CURSOR
cont>     FOR SELECT RESUME WHERE CURRENT OF TABLE_CURSOR;
SQL> --Open list cursor.
SQL> OPEN LIST_CURSOR;
SQL> --Load text from file into LIST OF BYTE VARYING column.
SQL> INSERT INTO CURSOR LIST_CURSOR
cont>     FILENAME 'shane_vilmer.resume' AS character varying;
62 segments inserted (maximum length 83)
SQL> CLOSE LIST_CURSOR;
SQL> CLOSE TABLE_CURSOR;
SQL> COMMIT;
```

INTEGRATE Statement

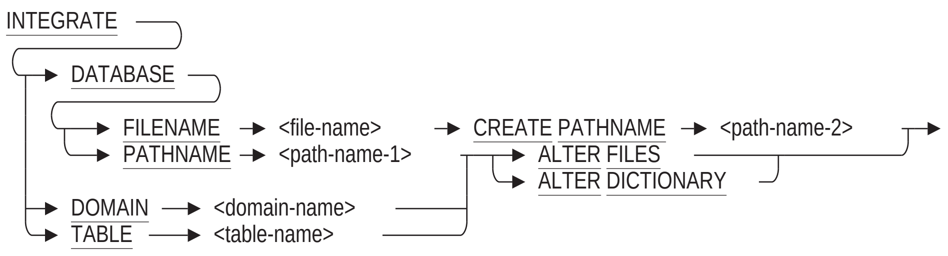
Makes definitions in a database and in a repository correspond by changing definitions in either the database or the repository.

The INTEGRATE statement can also create database definitions in the repository by copying from a database file to a specified repository.

Environment

You can issue the INTEGRATE statement only in interactive SQL.

Format



domain-name =

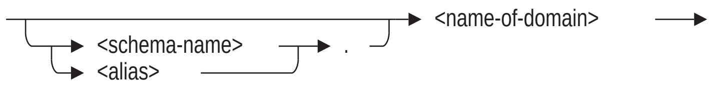
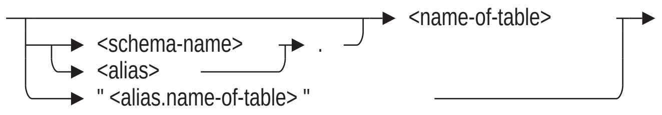


table-name =



Arguments

DATABASE FILENAME file-name CREATE PATHNAME path-name-2
Stores existing database system file definitions in the repository for the first time. See Example 8–3. 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.

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 8–1.

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.

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 8–2. Note, though, that altering database definitions in the repository 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.

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.

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.

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

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

- 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 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 Oracle CDD/Repository path names but valid Oracle Rdb domain and table names.

Examples

Example 8–1 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 8–1 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
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
$ !
$ ! 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 8–1 (Cont.) Updating the Database File Using Repository Definitions

```
$ !
$ ! Enter CDO again:
$ !
$ REPOSITORY
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
$ !
$ ! 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> !
```

(continued on next page)

Example 8–1 (Cont.) Updating the Database File Using Repository Definitions

```
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 8–2 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 8–2 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;
```

(continued on next page)

Example 8–2 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause

```
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

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
```

(continued on next page)

Example 8–2 (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
CDO> !
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> !
```

(continued on next page)

Example 8–2 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause

```
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
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 8–3 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 8–3 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 8–3 (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
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
$ !
$ ! Enter CDO again:
$ !
$ REPOSITORY
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 8–3 (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 8–4 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 8–4 Modifying Repository Field Using the INTEGRATE DOMAIN Statement with the ALTER DICTIONARY Clause

(continued on next page)

Example 8–4 (Cont.) 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 Oracle CDD/Repository
SQL> -- user interface and show the field DOMTEST from the TESTDB
SQL> -- database.
SQL> --
SQL> DISCONNECT ALL;
SQL> EXIT
$ CDO
CDO> SHOW FIELD DOMTEST FROM DATABASE TESTDB
Definition of field DOMTEST
|   Datatype                 text size is 5 characters
|   Collating sequence       'FRENCH'

CDO> !
CDO> ! Exit from Oracle 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
```

(continued on next page)

Example 8–4 (Cont.) Modifying Repository Field Using the INTEGRATE DOMAIN Statement with the ALTER DICTIONARY Clause

```
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 Oracle CDD/Repository V6.1
SQL> -- user interface and show the altered field DOMTEST from the TESTDB
SQL> -- database.
SQL> --
SQL> DISCONNECT ALL;
SQL> EXIT
$ CDO
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'
```

ITERATE Control Statement

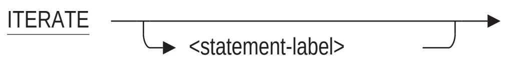
Causes the current iteration of the loop to abort and either the next iteration to start or the loop to terminate; depending on the termination conditions.

Environment

You can use the ITERATE 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



Arguments

statement-label
Names the label assigned to a compound statement or a loop statement.

Usage Notes

- The statement label must be for an active iterative loop statement. Iterative loop statements include LOOP, FOR cursor loop, FOR counted loop, WHILE, and REPEAT statements. An exception is raised if the specified label is unknown, not active, or is not a label for an iterative statement.
- If the statement label is omitted, then the innermost iterate statement is used by default. An exception is raised if there is no active iterative statement.

Example

Example 1: Using the ITERATE Control Statement

The following example shows the ITERATE control statement being used to prematurely complete the processing of the current row in a FOR cursor loop:

```
SQL> BEGIN
cont>   FOR :ord AS TABLE CURSOR ord_cursor
cont>   AS SELECT * FROM orders WHERE customer_id = :cid
cont>   DO
cont>      IF stock_count (:ord.product_id, :ord.quantity) IS NULL THEN
cont>           ITERATE;
cont>      END IF;
cont>      -- transfer stock to this order
cont>      UPDATE stock SET on_hand = on_hand - :ord.quantity
cont>       WHERE product_id = :ord.product_id;
cont>      UPDATE orders SET :ord.available = :ord.quantity
cont>       WHERE CURRENT OF ord_cursor;
cont>   END FOR;
cont> END;
```

LEAVE Control Statement

Unconditionally ends execution within a compound statement block or a looping statement but resumes execution on any SQL statement that immediately follows the exited statement.

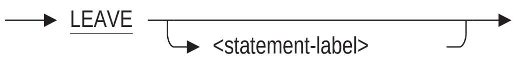
Environment

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 =



Arguments

statement-label
Names the label assigned to a compound statement, loop statement, or multistatement procedure.

Usage Notes

- The LEAVE statement can specify the name of the procedure if the compound statement it contains is not labeled. See Compound Statement for more information.
- If the statement-label is omitted, then the LEAVE statement leaves the currently active loop statement (WHILE, LOOP, REPEAT, FOR cursor loop, FOR counted loop); otherwise, it leaves the current labeled statement. If there is no active loop or labeled statement, then the current procedure is terminated.
- Do not use the LEAVE statement to leave SQL functions. A function must have a return result. You will receive a run-time error if you attempt to terminate a function with the LEAVE statement. Use the RETURN statement instead.

Examples

Example 1: Using the LEAVE control statement within a loop

```
SQL> set flags 'trace';
SQL>
SQL> create module REPORTS
cont> /*
***> This procedure counts the employees of a given state
***> who have had a decrease in their salary during their
***> employment
***> */
cont> procedure COUNT_DECREASED
cont>     (in :state CHAR(2)
cont>     ,inout :n_decreased INTEGER);
cont> begin
cont> set :n_decreased = 0;
cont>
cont> EMP_LOOP:
cont> for :empfor
cont>     as each row of
cont>         select employee_id
cont>          from EMPLOYEES where state = :state
cont> do
cont>     begin
cont>     declare :last_salary INTEGER (2) default 0;
cont>
cont>     HISTORY_LOOP:
cont>     for :salfor
cont>         as each row of
cont>             select salary_amount
cont>              from SALARY_HISTORY
cont>              where employee_id = :empfor.employee_id
cont>              order by salary_start
cont>     do
cont>         if :salfor.salary_amount < :last_salary
cont>         then
cont>             set :n_decreased = :n_decreased + 1;
cont>             trace :empfor.employee_id, ': ', :salfor.salary_amount;
cont>             leave HISTORY_LOOP;
cont>         end if;
cont>
cont>         set :last_salary = :salfor.salary_amount;
cont>     end for;
cont>     end;
cont> end for;
cont> end;
cont>
cont> end module;
SQL>
SQL> declare :n integer;
SQL> call COUNT_DECREASED ('NH', :n);
~Xt: 00200: 40789.00
~Xt: 00248: 46000.00
~Xt: 00471: 52000.00
           N
           3
SQL>
SQL> rollback;
```

Example 2: Ending Execution of a Compound Statement

```
PROCEDURE SAMPLE (IN :ID MONEY);
BEGIN
DECLARE: AMOUNT MONEY
         (SELECT TOTAL_AMOUNT FROM M_TABLE);
LOOP
   IF :AMOUNT IS NULL THEN
      LEAVE;
   END IF;
   .
   .
   .
   SET :AMOUNT =:AMOUNT-100.00;
   IF :AMOUNT < 0.00 THEN
       LEAVE;
   END IF;
END LOOP;
END;
```

LOCK TABLE Statement

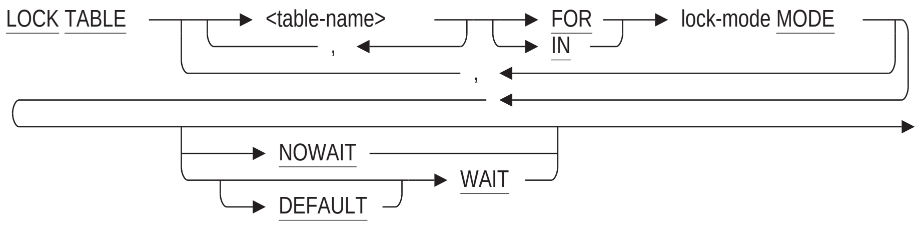
Specifies a list of tables to be readied in a given lock mode and added to the list of reserved tables for the current transaction. If a view is specified, then the base tables referenced by the view are locked in the specified lock mode.

Environment

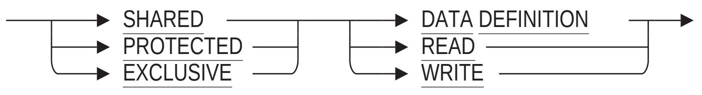
You can use the LOCK TABLE statement in a compound statement of a multistatement procedure:

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

Format



lock-mode =



Arguments

DATA DEFINITION
READ
WRITE
See the SET TRANSACTION statement for a description of these arguments.

IN lock-mode MODE
FOR lock-mode MODE
Specifies the lock mode to be used for the specified tables and views. The IN and FOR keywords are synonymous. A table lock mode can be promoted, but cannot be demoted. For example, you can promote a SHARED READ lock to SHARED WRITE, but you cannot demote a SHARED WRITE mode to a SHARED READ mode. See the Usage Notes for information on how the LOCK TABLE statement interacts with the SET TRANSACTION and DECLARE TRANSACTION statements.

SHARED
PROTECTED
EXCLUSIVE
See the SET TRANSACTION statement for a description of these arguments.

table-name
The names of one or more tables or views currently existing in the database that you want to lock and reserve. You can specify tables created as GLOBAL or LOCAL TEMPORARY TABLES, but they will be ignored because these types of tables do not contain shared data and so are never locked. You can specify tables from multiple databases by using the alias name as a prefix to the table name. If you do not specify an alias, then the default alias is used.

WAIT
NOWAIT
DEFAULT WAIT
Specifies what the LOCK TABLE statement does when it encounters a locked table. If you specify WAIT, the statement waits for other transactions to complete and then proceeds. If you specify NOWAIT, your transaction returns an error message when it encounters a locked table. If you specify DEFAULT WAIT, then the lock mode specified for the current transaction is used. If you specify a different lock mode than was specified for the transaction, the mode you specify with the LOCK TABLE statement takes precedence, unless the table is already reserved.

The WAIT clause is the default.

Usage Notes

- The LOCK TABLE statement has a definite advantage over the SET TRANSACTION RESERVING clause. It allows tables to be locked at modes other than SHARED READ when the table access is not determined until run time. For example, complex or dynamic applications often do not know the names of tables that will be accessed at the time a transaction is started. The LOCK TABLE statement allows those applications to start a transaction and add tables later, as they become known.
- If you start a transaction with a SET TRANSACTION or DECLARE TRANSACTION statement that includes the RESERVING clause, then all tables referenced during that transaction must have been specified in the reserving list of that transaction or subsequently with a LOCK TABLE statement. Exceptions to this rule are temporary tables and tables that are referenced by constraints and triggers. These tables are automatically reserved according to their access characteristics. For example, constraints require read access, triggers may require write access, and temporary tables require no special locking.
- If you start a transaction without specifying a list of reserved tables, then you can reference any tables during the transaction. By default, they will be accessed for SHARED READ or SHARED WRITE depending on the type of access statement issued. You can use the LOCK TABLE statement to adjust the default locking behavior as needed by the transaction.

- When you use multiple LOCK TABLE statements in a transaction, the tables can be reserved in any order and at any time, as you desire. However, this may lead to deadlocks in concurrent environments. Careful design can eliminate or minimize this problem. (Contrast this with the behavior seen when you use the SET TRANSACTION statement with the RESERVING clause. In this case, the tables are reserved using the order specified by the RDB$RELATION_ID column of the RDB$RELATION system relation so that a consistent ordering is used across every application. This avoids or eliminates deadlocks during table reservation.)
- If you issue a LOCK TABLE statement when no transaction is active, then a default transaction is started implicitly.
- The locks placed on tables by the LOCK TABLE statement are released when the transaction is terminated with a COMMIT, ROLLBACK, or DISCONNECT statement.

Examples

Example 1: Locking a Table in READ MODE

```
SQL> LOCK TABLE EMPLOYEES IN PROTECTED READ MODE NOWAIT;
```

Example 2: Locking Two Tables in Different Modes

```
SQL> LOCK TABLE DB1.JOB_HISTORY IN SHARED WRITE MODE,
cont>      DB2.SALARY_HISTORY IN EXCLUSIVE WRITE MODE;
```

LOOP Control Statement

Allows the repetitive execution of one or more SQL statements in a compound statement.

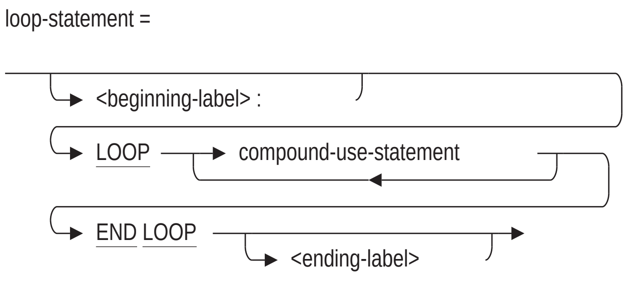
See also the FOR, REPEAT and WHILE statements.

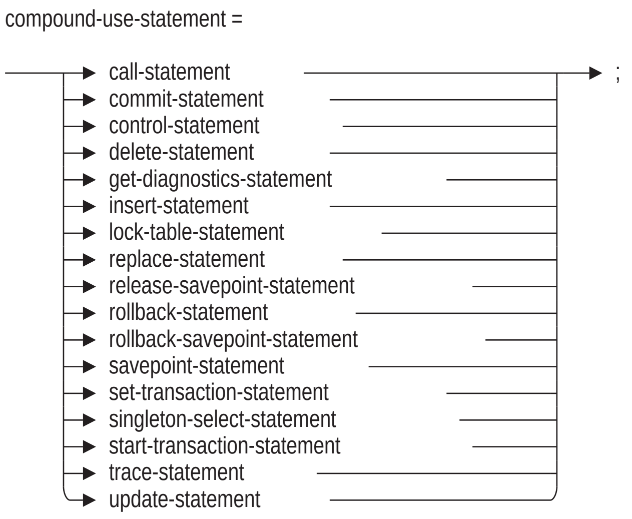
Environment

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





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.

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.

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.

Usage Note

LOOP will iterate indefinitely unless an exit condition is included.

Examples

Example 1: Executing a loop statement

```
SQL> create table ENROLLMENTS
cont>     (last_name         char(20),
cont>      first_name        char(10),
cont>      middle_initial    char,
cont>      class_name        char(10));
SQL>
SQL> begin
cont> declare :n integer default 5;
cont> loop
cont>     insert into ENROLLMENTS
cont>       values ('Jones', 'Robert', 'A',
cont>               'Class ' || CAST(:n as char(1)));
cont>     set :n = :n - 1;
cont>     if :n <= 0 then
cont>         leave;
cont>     end if;
cont> end loop;
cont> end;
SQL>
SQL> select * from ENROLLMENTS;
 LAST_NAME              FIRST_NAME   MIDDLE_INITIAL   CLASS_NAME
 Jones                  Robert       A                Class 5
 Jones                  Robert       A                Class 4
 Jones                  Robert       A                Class 3
 Jones                  Robert       A                Class 2
 Jones                  Robert       A                Class 1
5 rows selected
SQL>
```

OPEN Statement

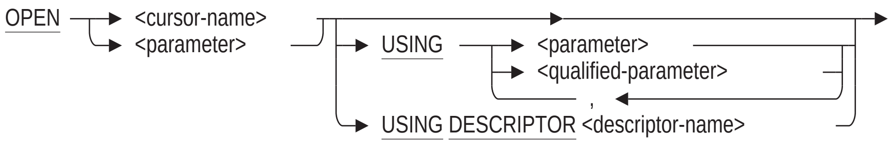
Opens a cursor so that rows of its result table can be retrieved through FETCH statements. The OPEN statement places the cursor before the first row of its result table.

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 Rdb 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

Example 1: Opening a cursor declared in a C or C++ 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
           AND D.DEPARTMENT_NAME CONTAINING :dept;
if (SQLCODE < 0) sql_signal ();

/* Open the cursor */
exec sql
    OPEN MANAGER;
if (SQLCODE < 0) sql_signal ();

/* Start a loop to process the rows of the cursor */
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;
    if (SQLCODE == 100) break;

    /* Print the values in the variables */
    printf ("%s %s %s\n", fname, lname, dname);
}

/* Close the cursor */
exec sql
    CLOSE MANAGER;
if (SQLCODE < 0) sql_signal ();
```

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 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 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 to execute the operating system command, you cannot use the dollar sign command to create logical names that affect the current interactive session. For instance, you cannot use the dollar sign command to change the value of the SQL$DATABASE logical.
- Interactive SQL interprets any command line that begins with a dollar sign ($) 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

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.
SQL>
```

PREPARE Statement

Prepares an SQL statement dynamically generated by a program for execution, and assigns a name to that 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.

The PREPARE . . . INTO statement stores in the SQLDA the number and data types of any select list items of a prepared statement. The SQLDA provides information about dynamic SQL statements to the program and information about memory allocated by the program to SQL.

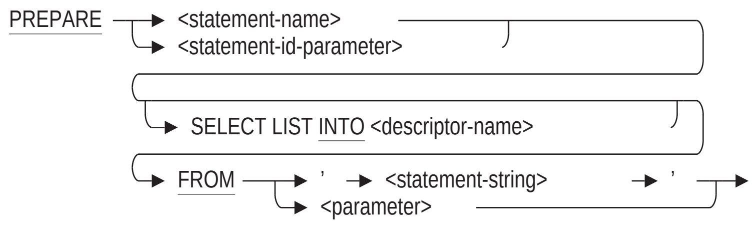
Appendix D describes in more detail the specific fields of the SQLDA, and how programs use it to communicate about select list items in prepared statements.

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

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 Appendix 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.
- 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 C. 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 the SQL HELP ERRORS.

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.

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 . . . INPUT 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.

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.

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.

Usage Notes

- The PREPARE statement sets values in the SQLCA to report the number of input and number of output parameters for a statement. These values allow memory to be allocated for input and output SQLDA structures.

  Assuming that the SQLERRD array is zero based, SQL sets SQLERRD[2] to the count of output parameters, and SQLERRD[3] to the count of input parameters. The values might be zero; CALL parameters of INOUT type will appear in both the input and output count. These values allow memory to be allocated for input and output SQLDA structures.

  These fields in the SQLCA were not set prior to Oracle Rdb release 7.1.3, therefore Oracle recommends that the SQLERRD[2] and SQLERRD[3] values be set to a known value (such as -1) prior to the PREPARE call. If the values remain as -1, the application must estimate the counts itself.
- 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 C.
- 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 . . . INPUT statement. To store information about parameter markers in the SQLDA, you must use the DESCRIBE . . . INPUT statement.

  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 C++. 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.

- If you use the statement-id-parameter, you will see one of the following behaviors:
  - If the statement-id is non-zero and does not match any prepared statement (the id was stale or contained a random value), then an error is raised:

    ```
    %SQL-F-BADPREPARE, Cannot use DESCRIBE or EXECUTE on a statement that is not prepared
    ```
  - If the statement-id is non-zero, or the statement name is one that has previously been used and matches an existing prepared statement, then that statement is automatically released prior to the prepare of the new statement. Refer to the RELEASE Statement for further details.
  - If the statement-id is zero or was automatically released, then a new statement-id is allocated and the statement prepared.

  If you use statement-name instead of a statement-id-parameter then SQL will implicitly declare an id for use by the application. Therefore, the semantics described apply similarly when using the statement-name. See the RELEASE Statement for details.
- 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

Table 8–1 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 8–1 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 Dynamic DECLARE CURSOR Extended dynamic DECLARE CURSOR DESCRIBE (optional) OPEN FETCH CLOSE RELEASE (optional)

DELETE
INSERT
UPDATE
SET statements
	Yes	No	PREPARE
			DESCRIBE (optional)
			EXECUTE
			EXECUTE IMMEDIATE (if no parameter markers)
			RELEASE (optional)

Compound statement
SELECT . . . INTO
INSERT . . .
RETURNING INTO
UPDATE . . .
RETURNING INTO
	Yes	Yes	PREPARE
			DESCRIBE (optional)
			EXECUTE
			EXECUTE IMMEDIATE (if no parameter markers)
			RELEASE (optional)

(continued on next page)

Table 8–1 (Cont.) SQL Statements That Can Be Dynamically Executed

Statement That Can Be Dynamically Executed	Parameter Markers Allowed?	Select List Items?	Associated Dynamic SQL Statements
ALTER ATTACH DECLARE TRANSACTION CREATE COMMENT ON COMMIT DROP GRANT RENAME REVOKE ROLLBACK SET TRANSACTION START TRANSACTION TRUNCATE	No	No	PREPARE EXECUTE EXECUTE IMMEDIATE RELEASE (optional)

Examples

Example 1: Preparing an INSERT statement with parameter markers

This C/C++ 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 stores the addresses of the variables in the SQLDA structure. 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.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sql_rdb_headers.h>

#define goto {\
   printf("sql error number=> %d\n", SQLCA.SQLCODE);\
   CheckSqlCode ();\
}
#define nowhere

exec sql WHENEVER SQLERROR   goto nowhere;
exec sql WHENEVER SQLWARNING goto nowhere;
/*
 *       Illustrate a dynamic INSERT statement
 *       with parameter markers:
 */

typedef struct emps_s {
    char emp_id[5];
    char fname[10];
    char lname[14];
    char city[20];
} emps_t;

char filespec[20],
     emp_id[5],
     fname[10],
     lname[14],
     city[20],
     command_string[256];

short emp_id_ind,
      fname_ind,
      lname_ind,
      city_ind;
/* Declare communication area (SQLCA) and descriptor area (SQLDA)
 * Define SQLVAR_ELEMENTS so that the structure can handle 10 elements
 */

#define SQLVAR_ELEMENTS 10
exec sql INCLUDE SQLDA;

exec sql INCLUDE SQLCA;

/* Declare the database: */
exec sql DECLARE ALIAS RUNTIME FILENAME :filespec;


void CheckSqlCode () {
    if (SQLCA.SQLCODE < 0) sql_signal ();
}

void Dump_s ()
{
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 the table:
 */

emps_t s_rec;

/* Declare indicator vector for the preceding structure: */

short s_ind[4];

printf ("\nDump the contents of S_REC\n");
```

```
exec sql OPEN X;
exec sql FETCH X INTO :s_rec indicator :s_ind;
while (SQLCA.SQLCODE == 0)
{
    printf ("\n%5d %s\n", s_ind[0], s_rec.emp_id);
    printf ("%5d %s\n", s_ind[1], s_rec.fname);
    printf ("%5d %s\n", s_ind[2], s_rec.lname);
    printf ("%5d %s\n", s_ind[3], s_rec.city);

    exec sql FETCH X INTO :s_rec indicator :s_ind;
}

exec sql CLOSE X;
return;
}

void main ()
{

printf ("Starting...\n");

/*
 * Assign values to FILESPEC and COMMAND_STRING,
 * and allocate memory for the SQLDA:
 */
strcpy (filespec, "SQL$DATABASE");
strcpy (command_string, "INSERT INTO EMPLOYEES \
        (EMPLOYEE_ID, FIRST_NAME, LAST_NAME, CITY) \
        VALUES (?,?,?,?)");
SQLDA = (struct SQLDA_STRUCT *) calloc (sizeof(struct SQLDA_STRUCT), 1);

SQLDA->SQLN = SQLVAR_ELEMENTS;

strncpy (&SQLDA->SQLDAID[0], "SQLDA   ",8);

/*
 * Prepare the statement assigned to COMMAND_STRING:
 */
exec sql PREPARE STMT3 FROM :command_string;
printf ("\nPrepared with %d input markers and %d output markers\n",
        SQLCA.SQLERRD[3], SQLCA.SQLERRD[2]);

/* Use a DESCRIBE statement to write information
 * about the parameter markers in the statement string
 * to the SQLDA:
 */
exec sql DESCRIBE STMT3 INPUT INTO SQLDA;

/* Assign values to the variables:
*/
strcpy (emp_id, "99999");
strcpy (fname, "Bob");
strcpy (lname, "Addams");
strcpy (city, "Francestown");

emp_id_ind = fname_ind = lname_ind = city_ind = 0;

/*
 * Assign the addresses of the variables to the SQLDATA field
 * of the SQLDA:
 */
SQLDA->SQLVAR[0].SQLDATA = (char *) &emp_id;
SQLDA->SQLVAR[1].SQLDATA = (char *) &fname;
SQLDA->SQLVAR[2].SQLDATA = (char *) &lname;
SQLDA->SQLVAR[3].SQLDATA = (char *) &city;

SQLDA->SQLVAR[0].SQLIND = &emp_id_ind;
SQLDA->SQLVAR[1].SQLIND = &fname_ind;
SQLDA->SQLVAR[2].SQLIND = &lname_ind;
SQLDA->SQLVAR[3].SQLIND = &city_ind;
```

```
/* 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:
 */
Dump_s ();

/*
 * Cleanup
 */
exec sql DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = '99999';
exec sql COMMIT WORK;

printf ("Done.\n");

}
```

Example 2: Showing the effect of the SQLCA support.

The following example program in C uses the PREPARE statement and then displays the SQLDA fields.

```
#include <stdio.h>
#include <sql_rdb_headers.h>

exec sql
    declare alias filename 'db$:mf_personnel';

exec sql
    include SQLCA;

char * s1 = "begin insert into work_status values (?, ?, ?);\
      select count(*) into ? from work_status; end";

void main ()
{
int i;
SQLCA.SQLERRD[2] = SQLCA.SQLERRD[3] = -1;
exec sql
    prepare stmt from :s1;

if (SQLCA.SQLCODE != 0) sql_signal ();

printf( "SQLCA:\n  SQLCODE:    %9d\n", SQLCA.SQLCODE);

for (i = 0; i < 6; i++)
    printf( "  SQLERRD[%d]: %9d\n", i, SQLCA.SQLERRD[i]);
}
```

The output shown below confirm that PREPARE processed three input arguments and one output argument.

```
SQLCA:
  SQLCODE:          0
  SQLERRD[0]:       0
  SQLERRD[1]:       0
  SQLERRD[2]:       1
  SQLERRD[3]:       3
  SQLERRD[4]:       0
  SQLERRD[5]:       0
```

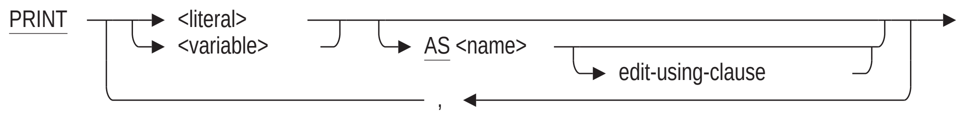
PRINT Statement

Displays a message in interactive SQL.

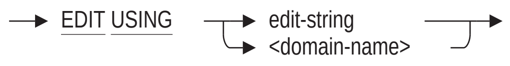
Environment

You can use the PRINT statement in interactive SQL.

Format



edit-using-clause =



Arguments

AS name
Changes the name displayed in the print statement header. By default literal values have a blank header name and variables use their name as a header. If the header must include spaces or lowercase characters then use SET QUOTING RULES or SET DIALECT to enable delimited identifiers

EDIT USING edit-string
EDIT USING domain-name
Assigns an edit string for use when formatting the variable or literal value. If a domain name is specified then the EDIT STRING from the domain is used.

This clause is only permitted for interactive SQL.

literal
Specifies the values you want displayed to the user during execution of the command procedure. Enclose the character literals in single quotation marks.

variable
Prints the contents 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. Use the SET OUTPUT statement to redirect the output to a file.

- If the variable was declared using a domain, then any EDIT STRING defined for the domain will be used by the PRINT statement to format the output.

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.

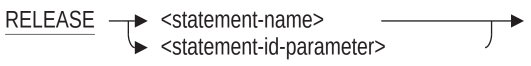
The RELEASE statement is a dynamic SQL statement. Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to SQL statements that are part of the source code for precompiled programs or SQL module language procedures. Unlike precompiled SQL or SQL module language statements, 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.
        .
        .
        .
```

RELEASE SAVEPOINT Statement

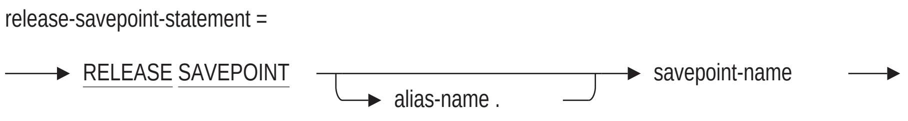
The RELEASE SAVEPOINT Statement destroys the named savepoint established by the SAVEPOINT statement. Changes made by the transaction are unaffected by this statement.

Environment

You can use the RELEASE SAVEPOINT statement in a compound statement of a multistatement procedure:

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

Format



Arguments

alias-name
This optional alias name can be used to target a specific database alias. If no alias-name is provided, then the current default database will be used.

savepoint-name
Name of a unique identifier for this savepoint. This name is declared using the SAVEPOINT statement.

Usage Notes

- If no established savepoint exists with this name, then the error RDB$_BAD_SVPT_HANDLE will be raised. SQLCODE will be returned as -882 and SQLSTATE will be returned as 3B001.

  ```
  %RDB-E-BAD_SVPT_HANDLE, invalid savepoint handle - "BOOKMARK2" is unknown
  ```
- The RELEASE SAVEPOINT statement may not be used in a SQL function definition, nor can it be called indirectly from a function.
- The RELEASE SAVEPOINT statement may not be called indirectly from a trigger action.

Examples

Example 1

The following example shows the use of the RELEASE SAVEPOINT statement.

```
SQL> set transaction read write;
SQL>
SQL> insert into module.SAMPLE values (1);
1 row inserted
SQL>
SQL> savepoint BOOK_IT;
SQL>
SQL> insert into module.SAMPLE values (2);
1 row inserted
SQL> insert into module.SAMPLE values (3);
1 row inserted
SQL>
SQL> table module.SAMPLE;
           A
           1
           2
           3
3 rows selected
SQL>
SQL> release savepoint BOOK_IT;
SQL>
SQL> insert into module.SAMPLE values (4);
1 row inserted
SQL>
SQL> table module.SAMPLE;
           A
           1
           2
           3
           4
4 rows selected
SQL>
SQL> commit;
SQL>
```

RENAME Statement

Allows the database administrator to change the name of a database object. This new name is then available for reference in other data definition statements, as well as from queries and routines.

Note

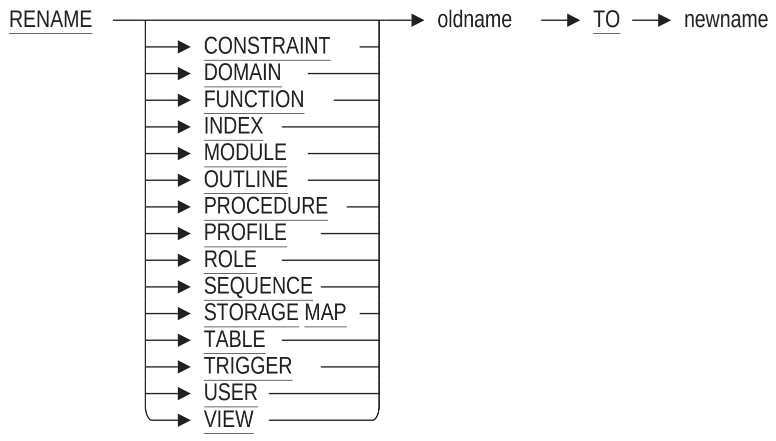
The RENAME statement may require that synonyms are enabled for the database. Reference the SYNONYMS ARE ENABLED clause of the ALTER, CREATE and IMPORT DATABASE statements.

Environment

You can use the RENAME statement:

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

Format



Arguments

newname
The new name for this object. This name must not already exist in the database for this object type, nor be the name of a synonym. The one exception is when the synonym references the **oldname** object. See the Usage Notes for further discussion.

If this is a RENAME TABLE, RENAME VIEW or RENAME SEQUENCE then the **newname** cannot be the name of an existing table, sequence or view.

oldname
The name of an existing object in the database. If the object type keyword is specified then an object must exist of that type. The name may also be a synonym for an object of the specified type.

Usage Notes

- You must have ALTER privilege on the database to rename a DOMAIN or OUTLINE.

  You must have ALTER privilege on the table, view, sequence, module, function or procedure to alter its name. If the procedure or function is part of a module then you will require only ALTER privilege on the containing module.

  You must have SECURITY privilege on the database to alter the name of a USER, ROLE or PROFILE.

  You must have ALTER privilege on the referencing table to rename a CONSTRAINT, or TRIGGER.
- The names of the database objects are stored in the Rdb system tables as both column values (for instance RDB$SEQUENCE_NAME) as well as encoded in binary definitions (such as RDB$VIEW_RSE) and original source (RDB$VIEW_SOURCE).

  The RENAME clause will modify all column values to reference the new name. However, the encoded values and original SQL source code are not modified by this command.

  To support these encoded definitions, as well as previously compiled applications, the old names are used to create synonyms that reference the new name of the object.

  The RENAME statement will create a synonym for the old names of domains, functions, modules, procedures, sequences, tables and views. These synonyms can be dropped if they are not used.

  Note

  It is not possible to create synonyms for OUTLINES, CONSTRAINTS, OUTLINES, PROFILES, ROLES, TRIGGERS, or USERS. Therefore, RENAME does not create synonyms for these objects. Care should be taken if the old names appear in module definitions, or application code.

- If a synonym already exists, and references the same object then it will be removed as part of the RENAME statement. For example, if you rename a table and wish to return to the previous **oldname** there will be an existing synonym with this name. Rdb will implicitly remove this synonym during the rename operation.
- The object type is optional. If no object type keyword is provided then Rdb will search for a matching name in this order:

  1. table or view
  2. domain
  3. function or procedure
  4. module

5. sequence
6. trigger
7. constraint
8. outline
9. user
10. role
11. profile
12. index
13. storage map

- When an IDENTITY column is created for a table, a sequence with the same name as the table is implicitly created. You may not use RENAME SEQUENCE on the identity sequence, use RENAME TABLE instead to alter the name of the table and its identity sequence.
- You may not RENAME an Rdb system table, index, storage map, view or sequence.
- RENAME INDEX changes the name of the index in all system tables.

  A synonym is created using the old index name to reference the new name of the index. This synonym will be used by any query outline that previously referenced the index using the old name. Note that only a single synonym name may exist. Therefore, if you have indices with the same name as another object, then the RENAME INDEX command may fail if creating the synonym detects a duplicate name.

  The command ALTER INDEX ... RENAME TO ... is synonymous with the RENAME INDEX command.
- RENAME STORAGE MAP changes the name of the storage map in all system tables.

  If the storage map has a companion function in the RDB$STORAGE_MAPS system module, then that function will also be renamed. A synonym is created using the old function name to reference the new name of the function. This synonym will be used by any other routine, computed by column, automatic column, and so on that referenced the old storage mapping function.

  The command ALTER STORAGE MAP ... RENAME TO ... is synonymous with the RENAME STORAGE MAP command.
- The following table compares the RENAME statements with the equivalent ALTER statements.

Table 8–2 Comparison between RENAME and ALTER Statements

RENAME statement	Equivalent ALTER statement	Is a synonym created?
RENAME CONSTRAINT	ALTER CONSTRAINT ... RENAME TO	No
RENAME DOMAIN	ALTER DOMAIN ... RENAME TO	Yes
RENAME FUNCTION	ALTER FUNCTION ... RENAME TO	Yes
RENAME INDEX	ALTER INDEX ... RENAME AS	Yes
RENAME MODULE	ALTER MODULE ... RENAME TO	Yes
RENAME OUTLINE	ALTER OUTLINE ... RENAME TO	No
RENAME PROCEDURE	ALTER PROCEDURE ... RENAME TO	Yes
RENAME PROFILE	ALTER PROFILE ... RENAME TO	No
RENAME ROLE	ALTER ROLE ... RENAME TO	No
RENAME SEQUENCE	ALTER SEQUENCE ... RENAME TO	Yes
RENAME STORAGE MAP	ALTER STORAGE MAP ... RENAME AS	Yes
RENAME TABLE	ALTER TABLE ... RENAME TO	Yes
RENAME TRIGGER	ALTER TRIGGER ... RENAME TO	No
RENAME USER	ALTER USER ... RENAME TO	No
RENAME VIEW	ALTER VIEW ... RENAME AS	Yes

Examples

Example 1: Preparing a database for RENAME statement

The RENAME statement for most objects requires that synonyms be enabled. This example shows the reported error if a RENAME is attempted for an object that requires synonyms.

```
SQL> attach 'filename personnel_sql';
SQL> show table
User tables in database with filename personnel_sql
     CANDIDATES
     COLLEGES
     CURRENT_INFO                    A view.
     CURRENT_JOB                     A view.
     CURRENT_SALARY                  A view.
     DEGREES
     DEPARTMENTS
     EMPLOYEES
     JOBS
     JOB_HISTORY
     RESUMES
     SALARY_HISTORY
     WORK_STATUS
SQL> rename table EMPLOYEES to COMPANY_STAFF;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-UNSSYNONYM, this database does not have synonyms enabled
SQL> disconnect all;
SQL> alter database filename personnel_sql synonyms are enabled;
```

Example 2: Renaming a table in the PERSONNEL database

This example renames the EMPLOYEES table. The SHOW TABLE statement lists the new name as well as synonym with the old name of the table.

```
SQL> attach 'filename personnel_sql';
SQL> rename table EMPLOYEES to COMPANY_STAFF;
SQL> show table
User tables in database with filename personnel_sql
     CANDIDATES
     COLLEGES
     COMPANY_STAFF
     CURRENT_INFO                    A view.
     CURRENT_JOB                     A view.
     CURRENT_SALARY                  A view.
     DEGREES
     DEPARTMENTS
     JOBS
     JOB_HISTORY
     RESUMES
     SALARY_HISTORY
     WORK_STATUS
     EMPLOYEES                       A synonym for table COMPANY_STAFF
SQL> select last_name from COMPANY_STAFF where employee_id = '00164';
 LAST_NAME
 Toliver
1 row selected
SQL>
```

Example 3: Renaming back to the original name

This example shows that the rename back to the original name will create a new synonym and remove the old synonym which had the same name as the tables new name.

```
SQL> rename table COMPANY_STAFF to EMPLOYEES;
SQL> show table
User tables in database with filename personnel_sql
     CANDIDATES
     COLLEGES
     CURRENT_INFO                    A view.
     CURRENT_JOB                     A view.
     CURRENT_SALARY                  A view.
     DEGREES
     DEPARTMENTS
     EMPLOYEES
     JOBS
     JOB_HISTORY
     RESUMES
     SALARY_HISTORY
     WORK_STATUS
     COMPANY_STAFF                   A synonym for table EMPLOYEES
SQL>
```

Example 4: Can not rename to a name used by the same object class or a synonym

The RENAME command does not allow the new name to be in use by the same class of objects, or by a synonym. In particular tables, views and sequences share the same name space.

```
SQL> rename view CURRENT_INFO to CURRENT_SALARY;
%SQL-F-REL_EXISTS, Table CURRENT_SALARY already exists in this database or
schema
SQL> create sequence CURRENT_INFORMATION;
SQL> rename view CURRENT_INFO to CURRENT_INFORMATION;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-SEQEXTS, there is another sequence named "CURRENT_INFORMATION" in
this database
-RDMS-F-RELNOTCHG, relation CURRENT_INFO has not been changed
```

Example 5: Using the RENAME INDEX and RENAME STORAGE MAP commands

```
SQL> show table (storage maps,index) employees
Information for table EMPLOYEES

Indexes on table EMPLOYEES:
EMPLOYEES_HASH                  with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Hashed Scattered
  Key suffix compression is DISABLED

EMP_EMPLOYEE_ID                 with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Sorted
  Key suffix compression is DISABLED
  Node size  430

EMP_LAST_NAME                   with column LAST_NAME
  Duplicates are allowed
  Type is Sorted
  Key suffix compression is DISABLED

Storage Map for table EMPLOYEES:
     EMPLOYEES_MAP

SQL> rename storage map EMPLOYEES_MAP to EMP_STORAGE_MAP;
SQL> rename index EMPLOYEES_HASH to EMP_ID_HASH;
SQL> show table (storage maps,index) employees
Information for table EMPLOYEES
```

```
Indexes on table EMPLOYEES:
EMP_EMPLOYEE_ID                 with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Sorted
  Key suffix compression is DISABLED
  Node size  430

EMP_ID_HASH                     with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Hashed Scattered
  Key suffix compression is DISABLED

EMP_LAST_NAME                   with column LAST_NAME
  Duplicates are allowed
  Type is Sorted
  Key suffix compression is DISABLED

Storage Map for table EMPLOYEES:
     EMP_STORAGE_MAP

SQL> show storage map
User Storage Maps in database with filename mf_personnel_sql
     CANDIDATES_MAP
     COLLEGES_MAP
     DEGREES_MAP
     DEPARTMENTS_MAP
     EMP_STORAGE_MAP
     JOBS_MAP
     JOB_HISTORY_MAP
     SALARY_HISTORY_MAP
     WORK_STATUS_MAP
SQL> show index
User indexes in database with filename mf_personnel_sql
     COLL_COLLEGE_CODE
     DEG_COLLEGE_CODE
     DEG_EMP_ID
     DEPARTMENTS_INDEX
     EMP_EMPLOYEE_ID
     EMP_ID_HASH
     EMP_LAST_NAME
     JH_EMPLOYEE_ID
     JOB_HISTORY_HASH
     SH_EMPLOYEE_ID
     EMPLOYEES_HASH                  A synonym for index EMP_ID_HASH
SQL> show system function
Functions in database with filename mf_personnel_sql
     CANDIDATES_MAP
     COLLEGES_MAP
     DEGREES_MAP
     DEPARTMENTS_MAP
     EMP_STORAGE_MAP
     JOBS_MAP
     JOB_HISTORY_MAP
     SALARY_HISTORY_MAP
     WORK_STATUS_MAP
     EMPLOYEES_MAP                   A synonym for function EMP_STORAGE_MAP
SQL>
```

REPEAT Control Statement

Repetitively executes one or more SQL statements in a compound loop until an end condition is met.

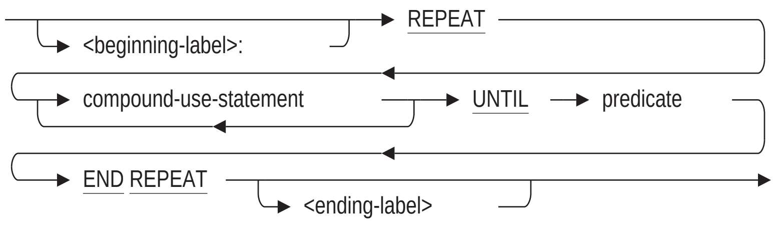
Environment

You can use the REPEAT 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

repeat-statement=



Arguments

beginning-label:
Assigns a name to the REPEAT statement. A beginning label used with the LEAVE statement lets you perform a controlled exit from a repeat loop. A named repeat loop is called a **labeled repeat loop statement**. A beginning label must be unique within the procedure in which the label is contained.

END REPEAT 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 ending-label argument makes multistatement procedures easier to read, especially in complex multistatement procedure blocks.

REPEAT compound-use-statement
Repeatedly executes a block of SQL statements until an end condition is met, as specified by the UNTIL predicate clause.

UNTIL predicate
Specifies a condition that controls how many times SQL can execute the statements embedded within its REPEAT . . . UNTIL block (collectively referred to as its compound statement). SQL executes the compound statement once and then evaluates the UNTIL condition. If it evaluates to false or NULL (unknown) and does not encounter an error exception, SQL executes the compound statement

again. Each time the search condition evaluates to false or NULL, the REPEAT statement executes the compound statement. If the UNTIL condition evaluates to true, SQL bypasses the compound statement and passes control to the statement after the END REPEAT statement.

Usage Notes

The loop body is executed at least once for a REPEAT statement.

Example

Example 1: Using a REPEAT Statement to List Files in the Current Directory

```
SQL> SET VERIFY;
SQL> ATTACH 'FILE SCRATCH';
SQL> CREATE DOMAIN file_name  VARCHAR(255);
SQL>  CREATE PROCEDURE find_file
cont>    (IN :FILESPEC file_name BY DESCRIPTOR,
cont>     INOUT :RESULTANT_FILESPEC file_name BY DESCRIPTOR,
cont>     INOUT :CONTEXT INTEGER BY REFERENCE);
cont> EXTERNAL NAME LIB$FIND_FILE
cont> LOCATION 'SYS$LIBRARY:LIBRTL.EXE'
cont> LANGUAGE GENERAL
cont> PARAMETER STYLE GENERAL
cont> COMMENT IS
cont>    'DCL HELP: LIB$FIND_FILE '
cont>     / 'The Find File routine is called with a wildcard  file'
cont>      / 'specification for  which   it   searches.    LIB$FIND_FILE '
SQL> CREATE PROCEDURE Find_file_end
cont>    (IN :CONTEXT INTEGER BY REFERENCE);
cont> EXTERNAL
cont>    NAME LIB$FIND_FILE_END
cont>     LOCATION 'SYS$LIBRARY:LIBRTL.EXE'
cont>     LANGUAGE GENERAL
cont>    PARAMETER STYLE GENERAL
cont> COMMENT IS
cont>   'DCL HELP: LIB$FIND_FILE_END '
cont>    / 'The End of Find File routine is called once'
cont>    / 'after each  sequence  of '
cont>    / 'calls to LIB$FIND_FILE. LIB$FIND_FILE_END deallocates'
cont>    / 'any saved Record Management Service (RMS) context and'
cont>    / 'deallocates the virtual memory used to hold the'
cont>    / 'allocated context block.';
```

```
SQL> SET FLAGS 'TRACE';
SQL> BEGIN
cont> -- This procedure performs a call to an external
cont> -- routine to list files located in the current
cont> -- default directory
cont> DECLARE :done, :context integer = 0;
cont> DECLARE :search_string FILE_NAME = '*.SQL';
cont> DECLARE :file_spec FILE_NAME;
cont> REPEAT
cont>     -- Ask the OpenVMS routine for the next name
cont>     CALL find_file (:search_string, :file_spec, :context);
cont>     IF POSITION ('*' in :file_spec) = 0
cont>        AND POSITION ('%' in :file_spec) = 0
cont>        AND POSITION ('...' in :file_spec) = 0
cont>     THEN
cont>         -- Display the name (there are no wildcards)
cont>         TRACE :file_spec;
cont>     ELSE
cont>        SET :done = 1;
cont>    END IF;
cont>   -- Exit when we have no more file names
cont>   UNTIL :done = 1
cont> END REPEAT;
cont> -- Clean up search context
cont> CALL find_file_end (:context);
cont> END;
~Xt: RDBVMS:[USER.V71]CREATE_ROLES.SQL;1
~Xt: RDBVMS:[USER.V71]TEST.SQL;1
SQL>
```

REPLACE Statement

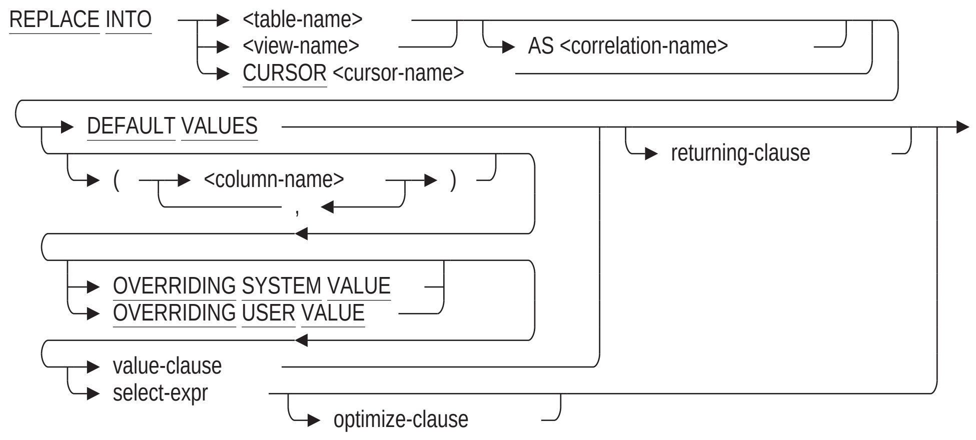
Replaces or adds a new row to a table or view. When a table includes a PRIMARY KEY definition, the REPLACE statement uses the key information to remove the existing matching row prior to inserting the replacement data.

Environment

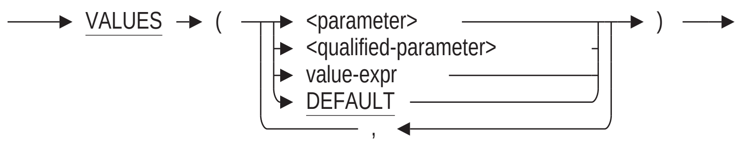
You can use the REPLACE statement:

- 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

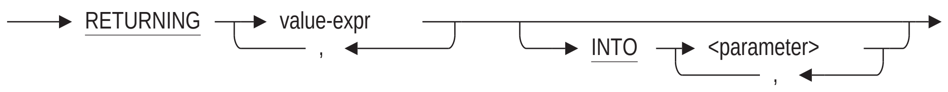
Format



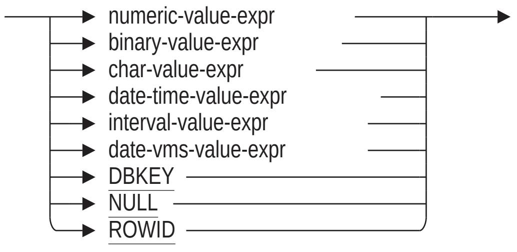
value-clause =



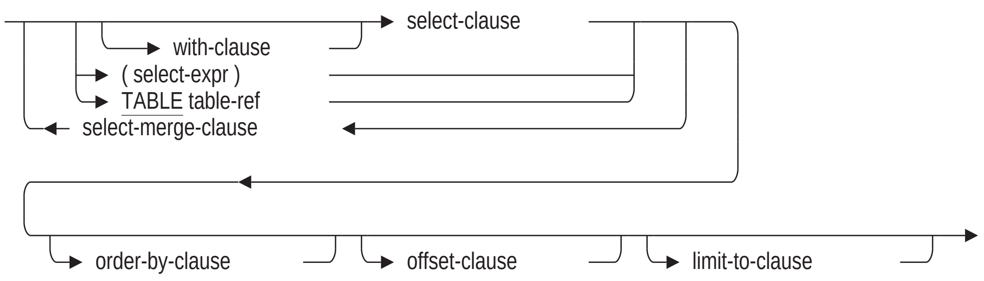
returning-clause=



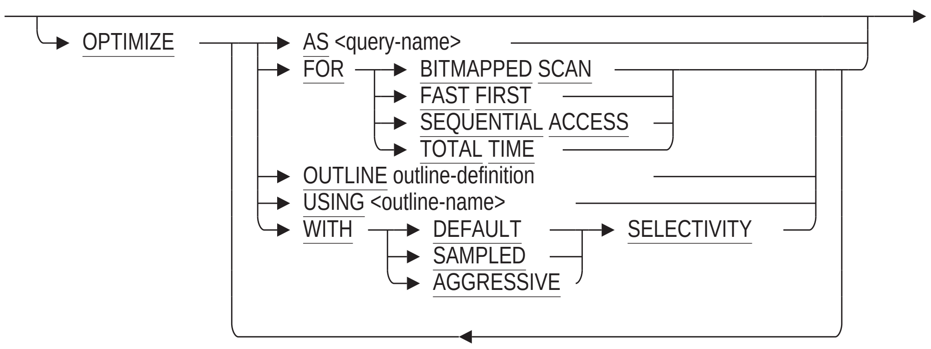
value-expr =



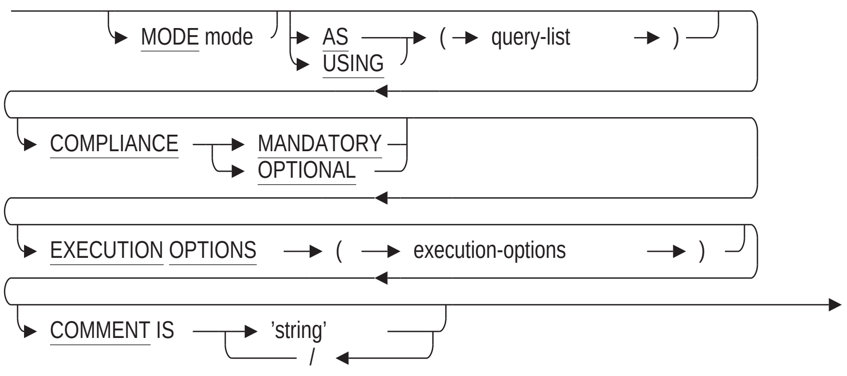
select-expr =



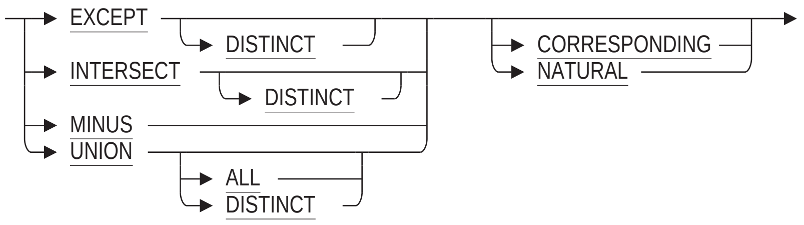
optimize-clause =



outline-definition =



select-merge-clause =



Arguments

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 as an AUTOMATIC INSERT AS column
- Defined with the IDENTITY attribute
- Defined with the NOT NULL clause or is part of a PRIMARY KEY clause of the CREATE TABLE statement

You cannot omit from a REPLACE statement the names of columns defined with the NOT NULL clause or PRIMARY KEY 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.

DEFAULT

Forces the named column to assume the default value defined for that column (or NULL if none is defined).

If the DEFAULT clause is used in an REPLACE statement then one of the following will be applied:

- If a DEFAULT attribute is present for the column then that value will be applied during REPLACE.
- If an AUTOMATIC INSERT clause is present for the column then that value will be applied during REPLACE. This can only happen if the SET FLAGS 'AUTO_OVERRIDE' is used since during normal processing these columns are read-only.
- Otherwise a NULL will be applied during REPLACE.

DEFAULT VALUES

Specifies that every column in the table be assigned a default value. Generated columns (such as AUTOMATIC INSERT AS, GENERATED, or IDENTITY) will be assigned the result of the value expression used to define the column. Otherwise NULL will be assigned for non-generated columns, or columns without a DEFAULT.

INTO parameter
Returns the values from the RETURNING clause into the specified list of parameters. There must exist one parameter for each value expression listed by the RETURNING clause.

This clause may be omitted for Interactive SQL which will then print the returned values.

INTO table-name
INTO view-name
The name of the target cursor, table or view to which you want to add a row.

limit-to-clause
See Section 2.8.1 for a description of the LIMIT TO expression.

OPTIMIZE AS query-name
The OPTIMIZE AS clause assigns a name to the query. Use the SET FLAGS 'STRATEGY' to see this name displayed.

OPTIMIZE FOR
The OPTIMIZE FOR clause specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- BITMAPPED SCAN

  Requests the Rdb query optimizer attempt to use BITMAPPED SCAN if there exists multiple supporting indices. This option is not compatible with SEQUENTIAL ACCESS.
- 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 canceled 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.

  Choose either FAST FIRST or TOTAL TIME. If optimization strategy is not explicitly set, FAST FIRST is the default.
- SEQUENTIAL ACCESS

  Forces the use of sequential access. This is particularly valuable for tables that use the strict partitioning functionality. This option is not compatible with BITMAPPED SCAN.
- 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.

  Choose either FAST FIRST or TOTAL TIME. Queries inside compound statements default to TOTAL TIME.

OPTIMIZE OUTLINE outline-definition
The OPTIMIZE OUTLINE clause declares a temporary query outline to be used with the select expression.

See the CREATE OUTLINE Statement for more information on defining an outline.

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 WITH
Selects one of three optimization controls: DEFAULT (as used by previous versions of Rdb), AGGRESSIVE (assumes smaller numbers of rows will be selected), and SAMPLED (which uses literals in the query to perform preliminary estimation on indices).

The following example shows how to use this clause.

```
SQL> select * from employees where employee_id  > '00200'
cont>   optimize with sampled selectivity;
```

order-by-clause
See Section 2.8.1 for a description of the ORDER BY expression.

OVERRIDING clause
The OVERRIDING SYSTEM VALUE clause instructs Rdb that the GENERATED, IDENTITY or AUTOMATIC AS columns will be replaced with user supplied values and therefore no generated values will be created. Such a clause would be used if a table was being reloaded after maintenance and the database administrator wanted to retain the saved generated values.

The OVERRIDING USER VALUE clause instructs Rdb that the GENERATED, IDENTITY or AUTOMATIC AS columns will be generated by the database system and that any user supplied values will be ignored. Such a clause would be used when column names were wild carded by a SELECT clause and therefore avoids enumerating all non-generated column names.

Without this clause REPLACE into AUTOMATIC, GENERATED (always) and IDENTITY (always) will result in a read-only column error.

RETURNING value-expr
Returns the values of columns from the target table. If DBKEY is specified, this argument returns the database key (dbkey) of the row being added. When the DBKEY value is valid, subsequent queries can use the DBKEY value to access the row directly. The ROWID keyword is a synonym for the DBKEY keyword.

select-clause
See Section 2.8.1 for a description of the SELECT expression.

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 updatable 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 updatable 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.

For detailed information on select expressions, see Section 2.8.1.

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.

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 updatable 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 updatable 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.

Values for IDENTITY, COMPUTED BY, and AUTOMATIC columns are not able to be replaced so these column types are not considered for the default column list.

See the SQL Online Help topic REPLACE EXAMPLES for an example that shows an REPLACE statement with a column select expression.

Usage Notes

- You must have both INSERT and DELETE privileges on the target table or view in order to replace rows.
- The REPLACE statement uses the PRIMARY KEY defined for the target table and erases any matching rows prior to inserting the new row contents.
- Since REPLACE can perform an implicit DELETE on the target table, Oracle Rdb will execute BEFORE and AFTER DELETE triggers, as well as BEFORE and AFTER INSERT triggers. There is no specific REPLACE trigger actions.
- If no PRIMARY KEY exists for the target table, or if one exists but is disabled then REPLACE acts just like an INSERT statement. In this case DELETE privilege is not required to insert into the table.
- When you use the REPLACE statement to replace rows from a view, you are actually replacing rows in the base table on which the view is based. In addition, SQL restricts the types of views with which you can use the REPLACE 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 replace rows from 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 replace rows into
SQL> -- it will fail because no value is provided for EMPLOYEE_ID:
SQL> REPLACE INTO TEMP (SUPERVISOR_ID) VALUES ('99999');
%RDB-E-INTEG_FAIL, violation of constraint JH_EMP_ID_EXISTS
caused operation to fail
-RDB-F-ON_DB, on database RDB$DEFAULT_CONNECTION
SQL> ROLLBACK;
SQL>
```

- To move data between databases, SQL lets you refer to a table from one database in the INTO clause of an REPLACE statement, and to tables from another database in a select expression within that REPLACE statement.

  REPLACE is one of the few SQL statements that allows you to specify a second database. Example 4 illustrates this point.

- Oracle Rdb uses the outline specified in the OPTIMIZE USING clause (if it exists) or as specified by the OPTIMIZE OUTLINE 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.

- If the target table of the REPLACE statement has an IDENTITY column then a new value will be generated for the replaced row.

  In addition the CURRVAL pseudo column can be used with the name of the table to reference this new sequence value. For instance, this example assumes the table ORDER has a column defined with the IDENTITY attribute.

```
SQL> replace into ORDER values (...);
SQL> replace into ORDER_LINES (ORDER.CURRVAL, ...);
SQL> replace into ORDER_LINES (ORDER.CURRVAL, ...);
```

This example shows that the FOREIGN KEY value is selected using a reference to the table name followed by the CURRVAL pseudo column.

However, the NEXTVAL pseudo column can not be used to fetch a new identity value. Only an INSERT or a REPLACE on the table can generate a new identity value.

- If the REPLACE on the table is rolled back or fails due to a constraint or trigger error condition, then the used identity values are discarded. If a row is deleted from the table, the identity value is not reused. For an exception to the reuse rule, see the usage note on TRUNCATE TABLE statement.

Examples

Example 1: Adding a row with literal values

The following interactive SQL example stores a replacement 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 62:765:9.

```
SQL> REPLACE 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
               62:765:9
1 row replaced
SQL>
```

Example 2: Adding a row using parameters

This example uses Interactive SQL to add 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 REPLACE operation if the value is less than zero.

- Uses the COMMIT statement to make the update permanent.

```
SQL> DECLARE :empl_id char(5);
SQL> DECLARE :job_code char(4);
SQL> DECLARE :start_date date vms;
SQL> DECLARE :end_date date vms;
SQL> DECLARE :end_date_ind integer;
SQL> DECLARE :dept_code char(4);
SQL> DECLARE :super char(5);
SQL>
SQL> ACCEPT :empl_id;
Enter value for EMPL_ID: 70001
SQL> ACCEPT :job_code UPPER;
Enter value for JOB_CODE: APGM
SQL> ACCEPT :start_date;
Enter value for START_DATE: 1-JUL-2010
SQL> ACCEPT :end_date :end_date_ind;
Enter value for END_DATE:  Exit
SQL> ACCEPT :dept_code UPPER;
Enter value for DEPT_CODE: PHRN
SQL> ACCEPT :super;
Enter value for SUPER: 00201
SQL>
SQL> SET TRANSACTION READ WRITE
cont>     RESERVING JOB_HISTORY FOR PROTECTED WRITE;
SQL>
SQL> REPLACE INTO JOB_HISTORY
cont>                (EMPLOYEE_ID,
cont>                 JOB_CODE,
cont>                 JOB_START,
cont>                 JOB_END,
cont>                 DEPARTMENT_CODE,
cont>                 SUPERVISOR_ID)
cont>         VALUES  (:EMPL_ID,
cont>                 :JOB_CODE,
cont>                 :START_DATE,
cont>                 :END_DATE :END_DATE_IND,
cont>                 :DEPT_CODE,
cont>                 :SUPER)
cont> ;
1 row replaced
SQL>
SQL> select EMPLOYEE_ID, JOB_CODE, JOB_START, JOB_END, DEPARTMENT_CODE
cont>  from JOB_HISTORY
cont>  where EMPLOYEE_ID = :empl_id;
 EMPLOYEE_ID   JOB_CODE   JOB_START     JOB_END       DEPARTMENT_CODE
 70001         APGM        1-Jul-2010   NULL          PHRN
1 row selected
SQL>
```

Example 3: Copying from one table to another

This interactive SQL example replaces data in 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 the city of ’Sanbornville’.

```
SQL>
SQL> REPLACE 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 CITY = 'Sanbornville';
1 row replaced
SQL>
```

Example 4: Copying rows between databases with the REPLACE statement

This example copies the contents of the EMPLOYEES table from the personnel database to another database, LOCALDATA.

```
SQL> ATTACH 'ALIAS LOCALDB FILENAME localdata';
SQL> ATTACH 'ALIAS PERS FILENAME personnel';
SQL>
SQL> SET 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> REPLACE INTO LOCALDB.EMPLOYEES
cont>   SELECT * FROM PERS.EMPLOYEES;
100 rows replaced
SQL>
```

Example 5: Using REPLACE to update changed row in a table.

```
SQL>
SQL> create table INVENTORY
cont>     (product_code integer
cont>         primary key not deferrable
cont>     ,product_refresh timestamp(2)
cont>     ,quantities integer
cont>     );
SQL>
SQL> create table IN_SHIPMENTS
cont>     like INVENTORY;
SQL>
```

During normal usage, the IN_SHIPMENTS table will be refreshed and applied to the INVENTORY table to replace on-hand quantities for products.

```
SQL>
SQL> -- Show the size of the inventory table before and after the REPLACE
SQL> select count(product_code) from INVENTORY;
```

```
                 1000
1 row selected
SQL>
SQL> -- Refresh the inventory with new updates
SQL> replace into INVENTORY
cont>     select product_code, current_timestamp, quantities
cont>     from IN_SHIPMENTS;
44 rows replaced
SQL>
SQL> select count(product_code) from INVENTORY;

                 1000
1 row selected
SQL>
```

Example 6: Replacing the user name and an amount into table columns:

```
SQL> CREATE TABLE TABLE1
cont>     (ID CHAR(31),
cont>      AMOUNT INT(4));
SQL>
SQL> REPLACE INTO TABLE1 (ID, AMOUNT)
cont>     VALUES (USER, 1000);
1 row replaced
SQL> SELECT * FROM TABLE1;
 ID                                          AMOUNT
 ELLINGSWORTH                             1000.0000
1 row selected
SQL>
```

Example 7: Replacing a name and a column select expression into the same table columns used in the previous example:

```
SQL> REPLACE INTO TABLE1 (ID, AMOUNT)
cont>     VALUES ('BROWN',
cont>       (SELECT COUNT (*) FROM TABLE1));
1 row replaced
SQL> SELECT * FROM TABLE1;
 ID                                          AMOUNT
 ELLINGSWORTH                             1000.0000
 BROWN                                       1.0000
2 rows selected
SQL>
```

Example 8: Replacing Default Values for Selected Columns

```
SQL> REPLACE INTO DEPARTMENTS
cont>     (DEPARTMENT_CODE, DEPARTMENT_NAME, BUDGET_ACTUAL)
cont>     VALUES
cont>     ('RECR','Recreation', DEFAULT);
1 row replaced
SQL>
SQL> SELECT DEPARTMENT_CODE, DEPARTMENT_NAME,
cont>       MANAGER_ID, BUDGET_PROJECTED
cont> FROM DEPARTMENTS WHERE DEPARTMENT_CODE='RECR';
 DEPARTMENT_CODE   DEPARTMENT_NAME   MANAGER_ID   BUDGET_PROJECTED
 RECR              Recreation        NULL                     NULL
1 row selected
SQL>
```

Example 9: Replacing a Row of All Default Values into a Table

This statement fails due to a PRIMARY KEY defined on the LAST_NAME column.

```
SQL> REPLACE INTO CANDIDATES
cont>     DEFAULT VALUES
cont> ;
%RDB-E-INTEG_FAIL, violation of constraint MUST_HAVE_NAME
caused operation to fail
-RDB-F-ON_DB, on database RDB$DEFAULT_CONNECTION
SQL>
SQL> SELECT LAST_NAME, FIRST_NAME FROM CANDIDATES
cont>     WHERE LAST_NAME IS NULL;
0 rows selected
SQL>
```

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

value-expr
The value expression to be returned as the result of this function call. The value-expr must be assignment-compatible with the data type defined by the stored function RETURNS clause.

See Section 2.6 for more information on value expressions.

Usage Notes

- The RETURN statement is required syntax when defining a stored function.
- If the RETURN statement is never executed, because of a conditional expression, then an exception is raised at run time.
- The RETURN statement is permitted only within stored functions.
- The RETURN statement should not be confused with the RETURNS clause of the stored function definition. The RETURNS clause defines the data type of the function, and the RETURN clause is executed to result the result.

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 Statements

Deletes privileges or roles from object access control.

Usage Notes

The following notes apply to all REVOKE statements.

- For the SELECT, INSERT, UPDATE 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.
- You cannot execute the REVOKE statement when any of the LIST, DEFAULT or RDB$SYSTEM storage areas are set to read-only. You must first set these storage areas to read/write. Note that in some databases RDB$SYSTEM will also be the default and list storage area.
- 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.
- You must execute the REVOKE statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.

REVOKE Statement

Removes privileges from or entirely deletes an entry in the Oracle Rdb access control list (ACL) for a database object. Each entry in an access control list consists of an identifier (or role) 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, procedure, function or sequence.

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.

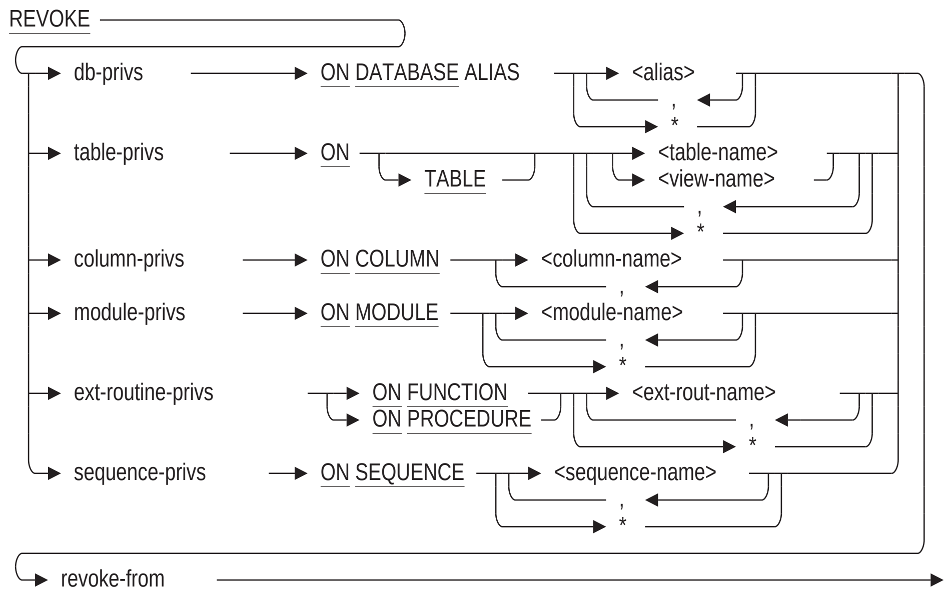
To create an entry or add privileges to an entry in the Oracle Rdb access control list for a database object, see the GRANT Statement.

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

Format


REVOKE
db-privs
ON DATABASE ALIAS
<alias>
,
*
table-privs
ON
TABLE
<table-name>
<view-name>
,
*
column-privs
ON COLUMN
<column-name>
,
module-privs
ON MODULE
<module-name>
,
*
ext-routine-privs
ON FUNCTION
ON PROCEDURE
<ext-rout-name>
,
*
sequence-privs
ON SEQUENCE
<sequence-name>
,
*
revoke-from

db-privs=

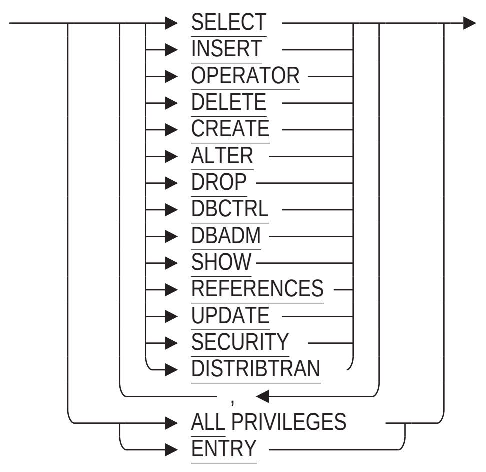
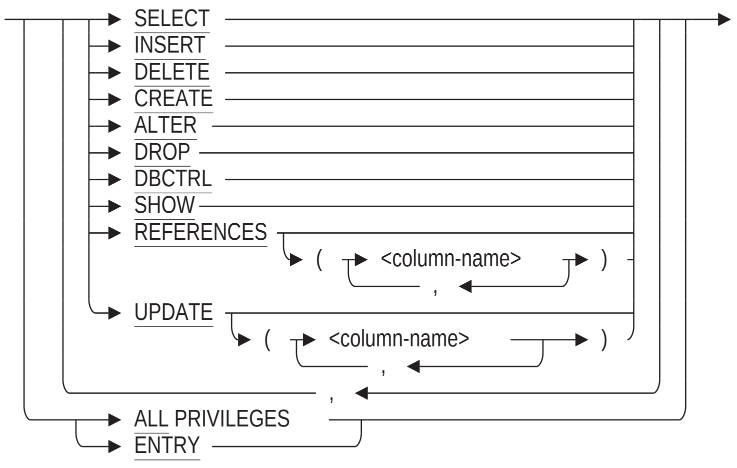
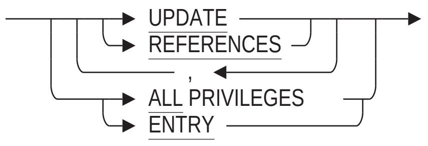

SELECT
INSERT
OPERATOR
DELETE
CREATE
ALTER
DROP
DBCTRL
DBADM
SHOW
REFERENCES
UPDATE
SECURITY
DISTRIBTRAN
,
ALL PRIVILEGES
ENTRY

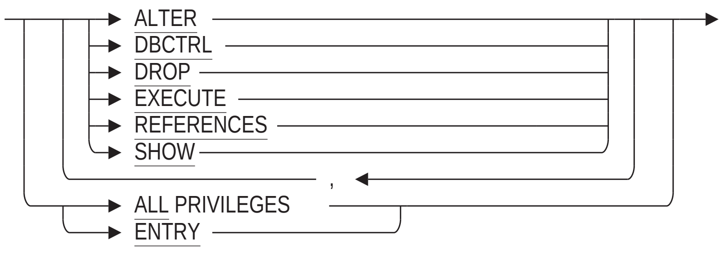
table-privs=



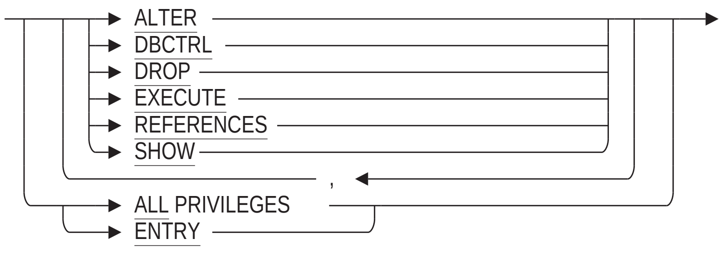
column-privs =



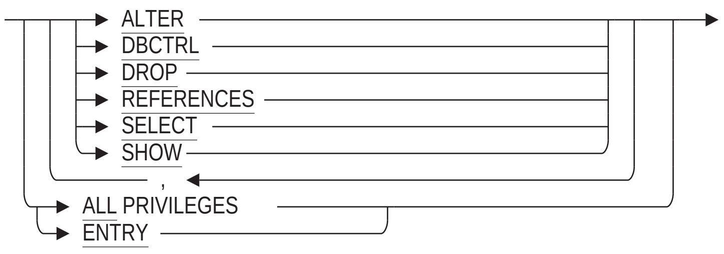
module-privs =



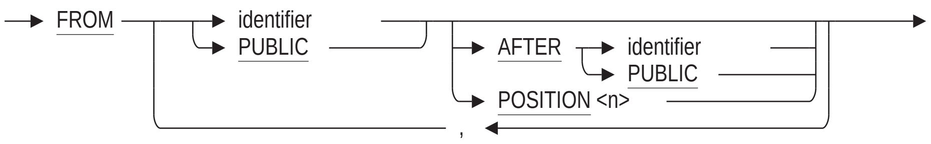
ext-routine-privs =



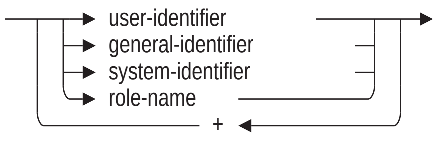
sequence-privs =



revoke-from =



identifier =



Arguments

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.

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).

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.

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 foyr types of identifiers:

- User identifiers
- General identifiers
- System-defined identifiers
- Role names

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.

For more information about identifiers, see your operating system documentation.

ON DATABASE ALIAS alias
ON TABLE table-name
ON COLUMN column-name
ON MODULE module-name
ON FUNCTION ext-routine-name
ON PROCEDURE ext-routine-name

ON SEQUENCE sequence-name
Specifies whether the REVOKE statement applies to ACLs for database objects. 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.

ON DATABASE ALIAS *
ON TABLE *
ON MODULE *
ON FUNCTION *
ON PROCEDURE *
ON SEQUENCE *
Specifies whether the REVOKE statement applies to ACLs for all objects of the specified types.

db-privs
table-privs
column-privs
module-privs
ext-routine-privs
sequence-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, external routine, or sequence. Table 7-5 in the GRANT Statement lists the privilege keywords and their meanings for databases, tables, modules, columns, external routines, and sequences.

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

role-name
The name of a role, such as one created with the CREATE ROLE statement. If the role name exists as an operating system group or rights identifier, then Oracle Rdb will create the role automatically when you issue the GRANT statement. A role that is created automatically always has the attribute of IDENTIFIED EXTERNALLY.

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

user-identifier
Uniquely identifies each user on the system.

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]

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.

Usage Notes

- You cannot REVOKE privileges on routines in a stored module; use REVOKE on the module instead.
- 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, external routine, or sequence 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
```

For more information on protection for an Oracle Rdb database, see the chapter on defining database privileges in the *Oracle Rdb Guide to Database Design and Definition*.

Example

Example 1: Using REVOKE to manage user access to the database and tables

```
SQL> attach 'filename DB$:MF_PERSONNEL';
SQL>
SQL> -- examine current privileges
SQL> show protection on database RDB$DBHANDLE;
Protection on Alias RDB$DBHANDLE
    (IDENTIFIER=SQLNET4RDB,ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+
      DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[DOC,DOC_READER],ACCESS=SELECT+CREATE)
    (IDENTIFIER=[DOC,DOC_WRITER],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
      ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+DROP+
      OPERATOR+DBADM+REFERENCES)
SQL>
SQL> -- revoke selected privileges
SQL> revoke CREATE on database alias RDB$DBHANDLE from DOC_WRITER;
SQL> revoke DISTRIBTRAN on database alias RDB$DBHANDLE from DOC_REVIEWER;
SQL> show protection on database RDB$DBHANDLE;
Protection on Alias RDB$DBHANDLE
    (IDENTIFIER=SQLNET4RDB,ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+
      DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
    (IDENTIFIER=[DOC,DOC_READER],ACCESS=SELECT)
    (IDENTIFIER=[DOC,DOC_WRITER],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+ALTER+
      DROP+DBCTRL+OPERATOR+DBADM+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+ALTER+DROP+
      OPERATOR+DBADM+REFERENCES)
SQL>
SQL> -- No longer all access to DOC_REVIEWER, use wildcard for all tables
SQL> revoke ALL PRIVILEGES on table * from DOC_REVIEWER;
SQL> commit;
```

Example 2: Revoking DROP Sequence Privileges from a User

```
SQL> CREATE SEQUENCE EMPID;
SQL> SHOW PROTECTION ON SEQUENCE EMPID
Protection on Sequence EMPID
    (IDENTIFIER=[RDB,STUART],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL)
    (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> GRANT SELECT ON SEQUENCE EMPID TO PUBLIC;
SQL> SHOW PROTECTION ON SEQUENCE EMPID;
Protection on Sequence EMPID
    (IDENTIFIER=[RDB,STUART],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL)
    (IDENTIFIER=[*,*],ACCESS=SELECT)
SQL> REVOKE DROP ON SEQUENCE EMPID FROM STUART;
SQL> SHOW PROTECTION ON SEQUENCE EMPID;
Protection on Sequence EMPID
    (IDENTIFIER=[RDB,STUART],ACCESS=SELECT+SHOW+ALTER+DBCTRL)
    (IDENTIFIER=[*,*],ACCESS=SELECT)
```

Example 3: REVOKE ALL versus REVOKE ENTRY

This example show the difference between the REVOKE ALL PRIVILEGES clause, which leaves the entry in tact but with no access, and the REVOKE ENTRY clause which removes the entire matching entry from the ACL (access control list).

```
SQL> create sequence next_dept;
SQL> grant all privileges on sequence next_dept to j_jones;
SQL> show prot on sequence next_dept;
Protection on Sequence NEXT_DEPT
    (IDENTIFIER=[RDB,J_JONES],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL+REFERENCES)
    (IDENTIFIER=[RDB,K_SMITH],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=NONE)
SQL>
SQL> ! Deny J_JONES all access to this sequence
SQL> revoke all privileges on sequence next_dept from j_jones;
SQL> show prot on sequence next_dept;
Protection on Sequence NEXT_DEPT
    (IDENTIFIER=[RDB,J_JONES],ACCESS=NONE)
    (IDENTIFIER=[RDB,K_SMITH],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=NONE)
SQL>
SQL> ! Remove the entry, J_JONES will now match only PUBLIC (aka [*,*])
SQL> revoke entry on sequence next_dept from j_jones;
SQL> show prot on sequence next_dept;
Protection on Sequence NEXT_DEPT
    (IDENTIFIER=[RDB,K_SMITH],ACCESS=SELECT+SHOW+ALTER+DROP+DBCTRL+REFERENCES)
    (IDENTIFIER=[*,*],ACCESS=NONE)
SQL>
```

REVOKE Statement: ANSI/ISO-Style

Removes privileges from the Oracle Rdb access control list granted to a specific user or role for a database object. 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, a role (role rights identifier) or the PUBLIC keyword.
- The set of privileges specifies what operations that user can perform on the database, table, column, module, procedure, function or sequence.

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 object and receives the privileges defined for PUBLIC. All privileges granted to held roles (role rights identifiers) are also merged with the access for the current user. A user without an entry in the access privilege set and who holds no matching roles receives only the privileges defined for PUBLIC. The PUBLIC identifier always has an entry in the access control list, even if PUBLIC has no access to the database object.

To create an entry or add privileges to an entry in the Oracle Rdb access control list for a a database object, see the GRANT Statement: ANSI/ISO-Style.

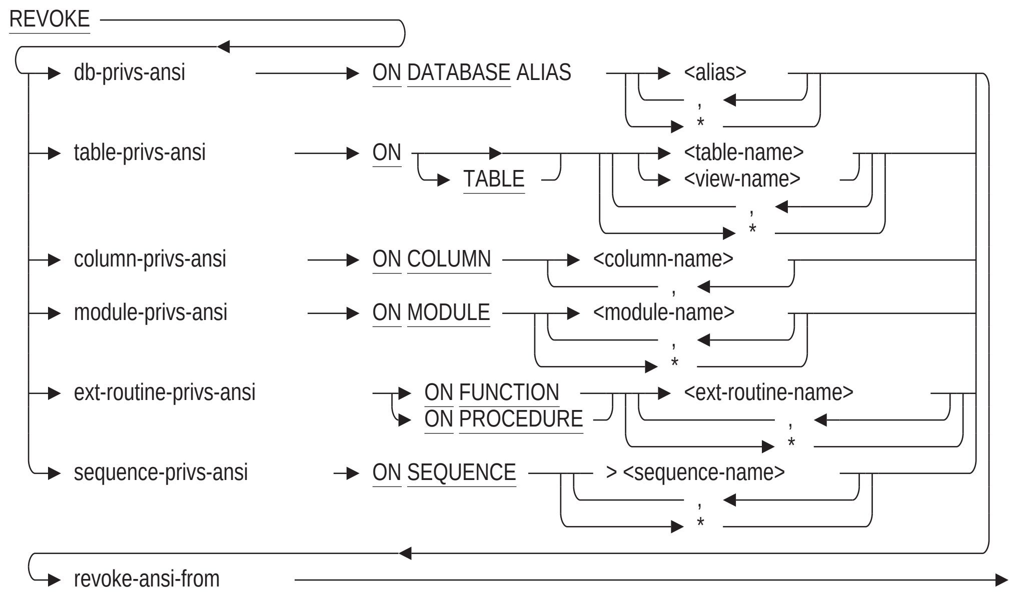
Environment

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

Format

REVOKE Statement: ANSI/ISO-Style



db-privs-ansi =

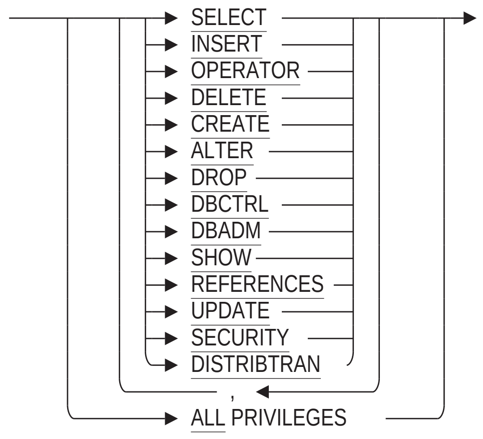
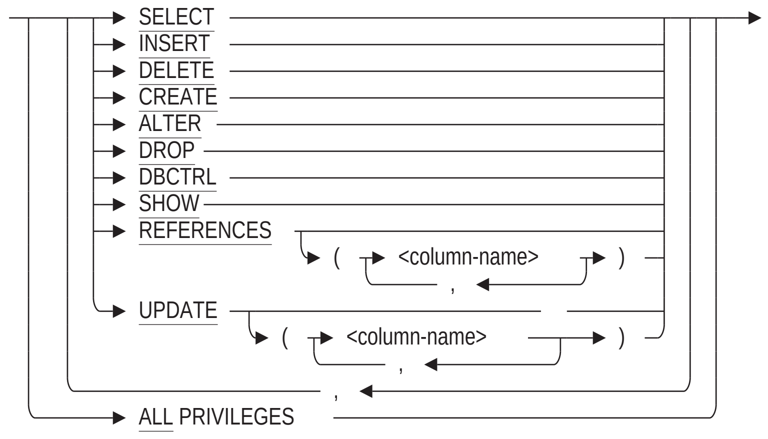
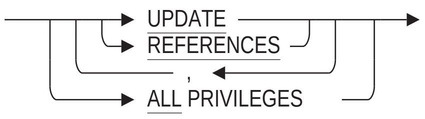


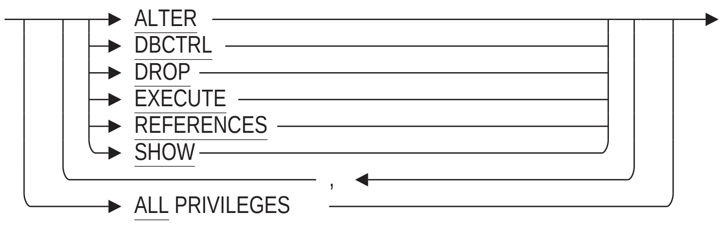
table-privs-ansi =



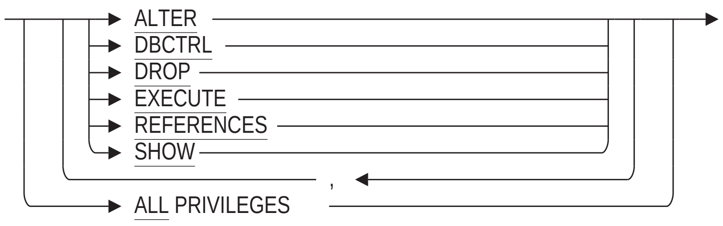
column-privs-ansi =



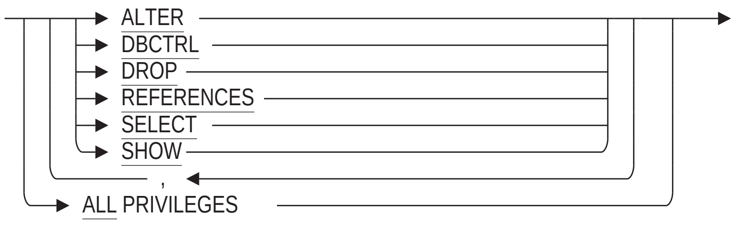
module-privs-ansi =



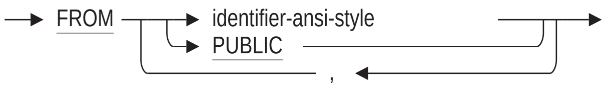
ext-routine-privs-ansi =



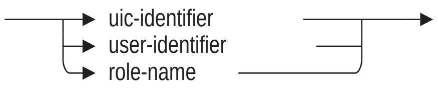
sequence-privs-ansi =



revoke-ansi-from =



identifier-ansi-style =



Arguments

ALL PRIVILEGES
Specifies that SQL should revoke all privileges in the access privilege set entry.

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).

For more information about user identifiers, see the operating system documentation.

ON DATABASE ALIAS alias
ON TABLE table-name
ON COLUMN column-name
ON MODULE module-name
ON FUNCTION ext-routine-name
ON PROCEDURE ext-routine-name
ON SEQUENCE sequence-name
Specifies whether the REVOKE statement applies to ACLs for database objects. 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.

ON DATABASE ALIAS *
ON TABLE *
ON MODULE *
ON FUNCTION *
ON PROCEDURE *
ON SEQUENCE *
Specifies whether the REVOKE statement applies to ACLs for all objects of the specified types. If privileges are denied for the operation on some objects, then the REVOKE is aborted.

db-privs-ansi
table-privs-ansi
column-privs-ansi
module-privs-ansi
ext-routine-privs-ansi
sequence-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, routine, or sequence. Table 7-5 in the GRANT Statement lists the privilege keywords and their meanings for databases, tables, modules, external routines and sequences.

TO identifier-ansi-style
Specifies the identifiers for the 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 single-user identifiers, roles (role rights identifiers), system and process rights; but no wildcard groups or members are allowed.

role-name
The name of a role, such as one created with the CREATE ROLE statement, or role rights identifier.

If the role name exists as an operating system group or rights identifier, then Oracle Rdb will create the role automatically when you issue the GRANT statement when SECURITY CHECKING IS INTERNAL. A role that is created automatically always has the attribute of IDENTIFIED EXTERNALLY.

user-identifier
Uniquely identifies each user on the system.

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]

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

- You can revoke only column-level privileges that have been specifically granted at the column level.
- To revoke the data manipulation privileges UPDATE and REFERENCES, you need to have been granted at least select 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 REVOKE privileges on routines in a stored module; use REVOKE on the module instead.

For more information on protection for an Oracle Rdb database, see the chapter on defining database privileges in the *Oracle Rdb Guide to Database Design and Definition*.

Examples

Example 1: Managing User Access with the REVOKE statement

```
SQL> attach 'filename DB$:ANSI_PERSONNEL';
SQL>
SQL> -- examine current privileges
SQL> show protection on database RDB$DBHANDLE;
```

```
Protection on Alias RDB$DBHANDLE
[DOC,DOC_WRITER]:
  With Grant Option:        SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,OPERATOR,DBADM,SECURITY,DISTRIBTRAN
  Without Grant Option:     SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,OPERATOR,DBADM,SECURITY,DISTRIBTRAN
[DOC,DOC_READER]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT,CREATE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
SQL>
SQL> -- revoke selected privileges
SQL> revoke CREATE on database alias RDB$DBHANDLE from DOC_READER;
SQL> revoke DISTRIBTRAN on database alias RDB$DBHANDLE from DOC_WRITER;
SQL> show protection on database RDB$DBHANDLE;
Protection on Alias RDB$DBHANDLE
[DOC,DOC_WRITER]:
  With Grant Option:        SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,OPERATOR,DBADM,SECURITY
  Without Grant Option:     SELECT,INSERT,UPDATE,DELETE,SHOW,CREATE,ALTER,DROP,
                            DBCTRL,OPERATOR,DBADM,SECURITY
[DOC,DOC_READER]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     NONE
SQL>
SQL> -- prevent drop by revoking the privilege
SQL> revoke DROP on table * from DOC_READER;
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
```

Example 4: Revoking DROP Privilege from a Sequence for a User

This example shows the action of REVOKE for a SEQUENCE in an ANSI style database.

```
SQL> create sequence EMPLOYEE_ID_GEN;
SQL> grant select on sequence EMPLOYEE_ID_GEN to public;
SQL> grant all privileges on sequence EMPLOYEE_ID_GEN to stuart;
SQL> show protection on sequence EMPLOYEE_ID_GEN;
Protection on Sequence EMPLOYEE_ID_GEN
[DOCS,STUART]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT,SHOW,ALTER,DROP,DBCTRL,REFERENCES
[DOCS,FREEMAN]:
  With Grant Option:        SELECT,SHOW,ALTER,DROP,DBCTRL,REFERENCES
  Without Grant Option:     NONE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
SQL> revoke drop on sequence EMPLOYEE_ID_GEN from stuart;
SQL> show protection on sequence EMPLOYEE_ID_GEN;
Protection on Sequence EMPLOYEE_ID_GEN
[DOCS,STUART]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT,SHOW,ALTER,DBCTRL,REFERENCES
[DOCS,FREEMAN]:
  With Grant Option:        SELECT,SHOW,ALTER,DROP,DBCTRL,REFERENCES
  Without Grant Option:     NONE
[*,*]:
  With Grant Option:        NONE
  Without Grant Option:     SELECT
SQL>
```

REVOKE Statement: Database System Privileges

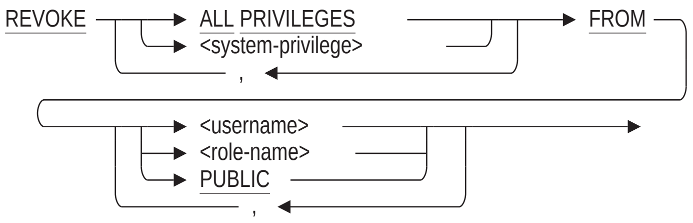
Revoke a Database system privilege from a user or role.

Environment

You can use the REVOKE statement for database system privileges:

- 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



Arguments

ALL PRIVILEGES
Revokes all system privileges assigned to the users and roles listed.

FROM username
FROM role-name
FROM PUBLIC
Specifies the user, role, or the PUBLIC user from which the specified system privilege is to be revoked.

system-privilege
The database system privileges to be revoked. Refer to the System Privileges table in the *GRANT Statement: Database System Privileges* for a list of supported system privileges.

Usage Notes

- You must have the SECURITY privilege on the database to revoke a role from a user or a role.

Example 8–5 Example 1: Revoking Database System Roles

```
SQL> show user J_JONES;
     J_JONES
     Identified externally
     Account is unlocked
     Granted Create Any
        VIEW
     Granted Alter Any
        VIEW
     Granted Drop Any
        VIEW
     No roles have been granted to this user
SQL>
SQL> -- This user is no longer responsible for managing views
SQL> revoke create any view, alter any view, drop any view from J_JONES;
SQL>
SQL> show user J_JONES;
     J_JONES
     Identified externally
     Account is unlocked
     No roles have been granted to this user
SQL>
```

Note that the REVOKE will take effect only for new sessions (after an ATTACH, CONNECT, etc).

Example

This example uses the REVOKE statement to remove those granted system privileges. If all the database system privileges are to be removed, then the ALL PRIVILEGES clause could have been used.

REVOKE Statement: Roles

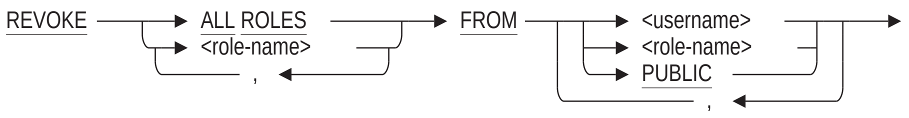
Revoke a role from another user or role.

Environment

You can use the REVOKE statement for roles:

- 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



Arguments

ALL ROLES
Revokes all roles assigned to the users listed.

FROM username
FROM role-name
FROM PUBLIC
Specifies the user, role, or the PUBLIC user from which the specified role is to be revoked.

role-name
The name of an existing role created with the CREATE ROLE statement or created automatically by the GRANT statement.

Usage Notes

- You must have the SECURITY privilege on the database to revoke a role from a user or another role.

Example

Example 1: Granting and Revoking Roles

```
SQL> -- Optionally, create three users and two roles.
SQL> -- Oracle Rdb automatically generates users and
SQL> -- roles if they are identified externally.
SQL> CREATE USER ABLOWNEY IDENTIFIED EXTERNALLY;
SQL> CREATE USER BGREMBO IDENTIFIED EXTERNALLY;
SQL> CREATE USER LWARD IDENTIFIED EXTERNALLY;
SQL> CREATE ROLE SALES_MANAGER IDENTIFIED EXTERNALLY;
SQL> CREATE ROLE DIVISION_MANAGER IDENTIFIED EXTERNALLY;
SQL> -- Grant the SALES_MANAGER role to users ABLOWNEY and
SQL> -- BGREMBO. Also grant the SALES_MANAGER role to the
SQL> -- DIVISION MANAGER ROLE.
SQL> GRANT SALES_MANAGER TO ABLOWNEY, BGREMBO, DIVISION_MANAGER;
SQL> -- Grant the DIVISION_MANAGER role to LWARD. LWARD now
SQL> -- has both the SALES_MANAGER and DIVISION_MANAGER roles.
SQL> GRANT DIVISION_MANAGER TO LWARD;
SQL> -- Revoke the DIVISION_MANAGER role from LWARD. He has
SQL> -- left the company.
SQL> REVOKE DIVISION_MANAGER FROM LWARD;
SQL> -- Grant the DIVISION_MANAGER role to BGREMBO.  She
SQL> -- has been promoted to division manager.
SQL> GRANT DIVISION_MANAGER TO BGREMBO;
```

ROLLBACK Statement

Ends a transaction and undoes all changes you made since that transaction began. The ROLLBACK statement also:

- Closes all open cursors (with the exception of WITH HOLD cursors)
- Releases all row locks
- Performs a checkpoint operation if fast commit processing is enabled

The ROLLBACK statement affects:

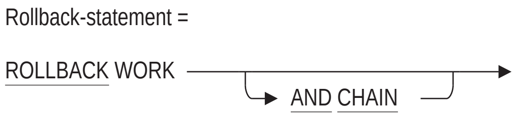
- All open databases included in the current transaction
- All changes to data made with SQL data manipulation statements (DELETE, UPDATE, and INSERT)
- All changes to data definitions made with SQL data definition statements (ALTER, CREATE, DROP, RENAME, GRANT, and REVOKE)

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



Arguments

AND CHAIN
Starts a new transaction implicitly using the same attributes as this rolled back transaction.

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.
- The ROLLBACK statement may not be executed from a SQL function or trigger or any stored procedure called from a SQL function or trigger.

- The AND CHAIN clause is only permitted in a compound statement (i.e. in a BEGIN . . . END block), or as the body of a stored procedure.
- When AND CHAIN is used a new transaction is implicitly started using the same attributes as the rolled back transaction. Attributes such as READ WRITE, READ ONLY, RESERVING, EVALUATING, WAIT, and ISOLATION LEVEL are retained for the new transaction.
- Applications can use the AND CHAIN clause to simplify applications, since the complex transaction attributes need only be specified once.
- When the SET FLAGS option TRANSACTION_PARAMETERS is specified a line of output is written to identify the rolled-back and chained transaction. Each SET TRANSACTION assigns a unique sequence number which is displayed after each transaction action line.
- When the ROLLBACK statement is executed within a compound statement and no transaction is active, a success status (SQLSTATE or SQLCODE) is the result.

  However, if the ROLLBACK statement is executed in a single statement, it will result in an error. This behavior can be modified by setting the dialect to SQL92 or SQL99, or by using the SET QUIET COMMIT statement. Refer to the SET DIALECT and SET QUIET COMMIT statements for more details. For SQL Module Language or SQL pre-compiler applications, refer to the QUIET_COMMIT qualifier and the QUIET COMMIT clause in the module header.

Examples

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.
```

Example 2: Using COMMIT and AND CHAIN

The following simple example executes SET TRANSACTION once at the start of the procedure. Then periodically the transaction is committed and restarted using the COMMIT AND CHAIN syntax. This simplifies the application since there is only one definition of the transaction characteristics.

```
SQL> -- process table in batches
SQL>
SQL> set compound transactions 'internal';
SQL> set flags 'transaction,trace';
SQL>
SQL> begin
cont> declare :counter integer = 0;
cont> declare :emp integer;
cont>
cont> set transaction
cont>     read write
cont>     reserving employees for exclusive write;
cont>
cont> for :emp in 0 to 600
cont> do
cont>     begin
cont>     declare :id char(5)
cont>           default substring (cast (:emp+100000 as varchar(6))
cont>                              from 2 for 5);
cont>     if exists (select * from employees where employee_id = :id)
cont>     then
cont>         trace 'found: ', :id;
cont>         if :counter > 20
cont>         then
cont>             commit and chain;
cont>             set :counter = 1;
cont>         else
cont>             set :counter = :counter + 1;
cont>         end if;
cont>     end if;
cont>     end;
cont> end for;
cont>
cont> commit;
cont> end;
~T Compile transaction (1) on db: 1
~T Transaction Parameter Block: (len=2)
0000 (00000) TPB$K_VERSION = 1
0001 (00001) TPB$K_WRITE (read write)
~T Start_transaction (1) on db: 1, db count=1
~T Rollback_transaction on db: 1
~T Compile transaction (3) on db: 1
~T Transaction Parameter Block: (len=14)
0000 (00000) TPB$K_VERSION = 1
0001 (00001) TPB$K_WRITE (read write)
0002 (00002) TPB$K_LOCK_WRITE (reserving) "EMPLOYEES" TPB$K_EXCLUSIVE
~T Start_transaction (3) on db: 1, db count=1
~Xt: found: 00164
   .
   .
   .
~Xt: found: 00184
~Xt: found: 00185
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
~T Restart_transaction (3) on db: 1, db count=1
~Xt: found: 00186
   .
   .
   .
```

```
~Xt: found: 00205
~Xt: found: 00206
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
~T Restart_transaction (3) on db: 1, db count=1
~Xt: found: 00207
   .
   .
   .
~Xt: found: 00228
~Xt: found: 00229
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
~T Restart_transaction (3) on db: 1, db count=1
~Xt: found: 00230
   .
   .
   .
~Xt: found: 00249
~Xt: found: 00267
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
~T Restart_transaction (3) on db: 1, db count=1
~Xt: found: 00276
   .
   .
   .
~Xt: found: 00435
~Xt: found: 00471
~T Commit_transaction on db: 1
~T Prepare_transaction on db: 1
SQL>
```

ROLLBACK TO SAVEPOINT Statement

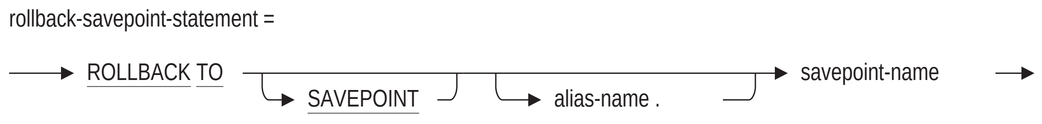
The ROLLBACK TO SAVEPOINT statement destroys the named savepoint established by the SAVEPOINT statement and removes all database changes made from the time the SAVEPOINT statement established the named savepoint.

Environment

You can use the ROLLBACK TO SAVEPOINT statement in a compound statement of a multistatement procedure:

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

Format



Arguments

alias-name
This optional alias name can be used to target a specific database alias. If no alias-name is provided, then the current default database will be used.

savepoint-name
Name of a unique identifier for this savepoint. This name is declared using the SAVEPOINT statement.

Usage Notes

- If no established savepoint exists with this name, then the error RDB$_BAD_SVPT_HANDLE will be raised. SQLCODE will be returned as -882 and SQLSTATE will be returned as 3B001.

  ```
  %RDB-E-BAD_SVPT_HANDLE, invalid savepoint handle - "BOOKMARK2" is unknown
  ```
- The ROLLBACK TO SAVEPOINT statement may not be used in a SQL function definition, nor can it be called indirectly from a function.
- The ROLLBACK TO SAVEPOINT statement may not be called indirectly from a trigger action.

Examples

Example 1

The following example shows the use of SAVEPOINT and ROLLBACK TO SAVEPOINT to exclude rows inserted during the transaction. In an actual application, the ROLLBACK TO SAVEPOINT statement would probably be within a conditional statement such as IF-THEN-ELSE or CASE statement.

```
SQL> declare local temporary table module.SAMPLE
cont>     (a integer)
cont>     on commit preserve rows
cont> ;
SQL>
SQL> set transaction read only;
SQL>
SQL> insert into module.SAMPLE values (1);
1 row inserted
SQL>
SQL> savepoint BOOK_IT;
SQL>
SQL> insert into module.SAMPLE values (2);
1 row inserted
SQL> insert into module.SAMPLE values (3);
1 row inserted
SQL>
SQL> table module.SAMPLE;
           A
           1
           2
           3
3 rows selected
SQL>
SQL> rollback to savepoint BOOK_IT;
SQL>
SQL> insert into module.SAMPLE values (4);
1 row inserted
SQL>
SQL> table module.SAMPLE;
           A
           1
           4
2 rows selected
SQL>
SQL> commit;
SQL>
```

SAVEPOINT Statement

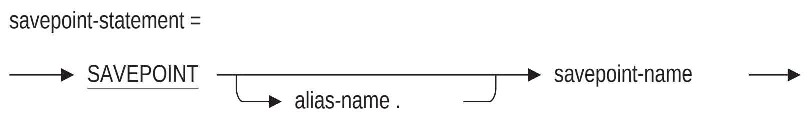
The SAVEPOINT Statement establishes a marker in the current transaction that allows the programmer to undo part of the transaction (using ROLLBACK TO SAVEPOINT) without resorting to a full transaction ROLLBACK.

Environment

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

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

Format



Arguments

alias-name
This optional alias name can be used to target a specific database alias. If no alias-name is provided, then the current default database will be used.

savepoint-name
Name of a unique identifier for this savepoint. This name will be used with subsequent ROLLBACK TO SAVEPOINT and RELEASE SAVEPOINT statements.

Usage Notes

- If the SAVEPOINT statement is used more than once with the same name, then the prior SAVEPOINT is destroyed and replaced with this new location.
- Any established savepoints will be discarded by a ROLLBACK statement (which does not use the TO SAVEPOINT clause), and by a COMMIT statement.
- If more savepoints are created than are supported by Rdb, then the error RDB$_EXCESS_SVPT will be raised. SQLCODE will be returned as -880 and SQLSTATE will be returned as 3B002.

```
%RDB-E-EXCESS_SVPT, maximum number of savepoints are already active - "BOOK2"
failed
```

- The SAVEPOINT statement may not be used in a SQL function definition, nor can it be called indirectly from a function.

- The SAVEPOINT statement may not be called indirectly from a trigger action.
- A SAVEPOINT statement is only valid if a transaction is in progress. This can be either a READ WRITE or READ ONLY transaction. Note that temporary tables can be updated during a read only transaction.

```
SQL> commit;
SQL> savepoint BK;
%RDB-E-NOTXNINPRGS, no transaction is in progress
-RDB-E-SVPT_NOALLOWED, a savepoint may not be established in this context -
"BK" failed
```

Examples

Example 1

The following example shows the use of the SAVEPOINT statement. Note that reusing the savepoint name will re-establish that marker and so affect different rows in the transaction.

```
SQL> declare local temporary table module.SAMPLE
cont>     (a integer)
cont>     on commit preserve rows
cont> ;
SQL>
SQL> --
SQL>
SQL> set transaction read only;
SQL>
SQL> insert into module.SAMPLE values (1);
1 row inserted
SQL>
SQL> -- Establish the initial marker
SQL> savepoint BOOK_IT;
SQL>
SQL> insert into module.SAMPLE values (2);
1 row inserted
SQL> insert into module.SAMPLE values (3);
1 row inserted
SQL>
SQL> table module.SAMPLE;
           A
           1
           2
           3
3 rows selected
SQL>
SQL> -- Move the marker
SQL> savepoint BOOK_IT;
SQL>
SQL> insert into module.SAMPLE values (4);
1 row inserted
SQL>
SQL> rollback to savepoint BOOK_IT;
SQL>
SQL> table module.SAMPLE;
           A
           1
           2
           3
3 rows selected
SQL>
SQL> commit;
SQL>
```

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.

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:

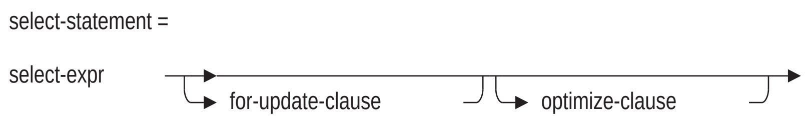
1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. Select list
6. ORDER BY
7. OFFSET
8. LIMIT TO (or FETCH FIRST)
9. OPTIMIZE

After each of these clauses, SQL produces an intermediate result table that is used in evaluating the next clause.

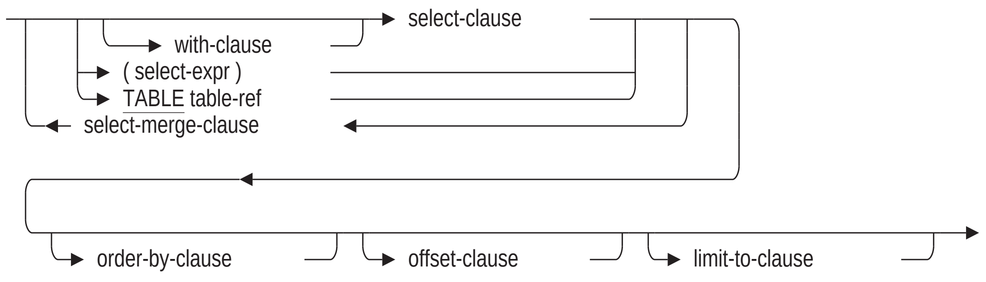
Environment

You can use the general form of the SELECT statement only in interactive and dynamic SQL.

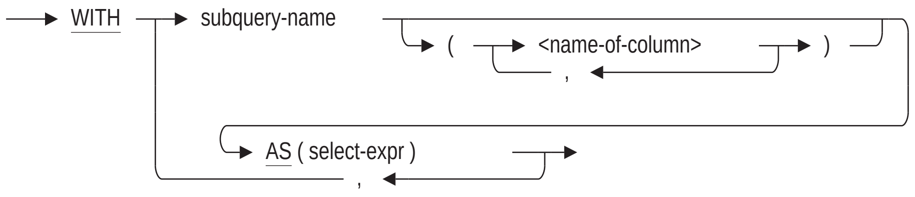
Format



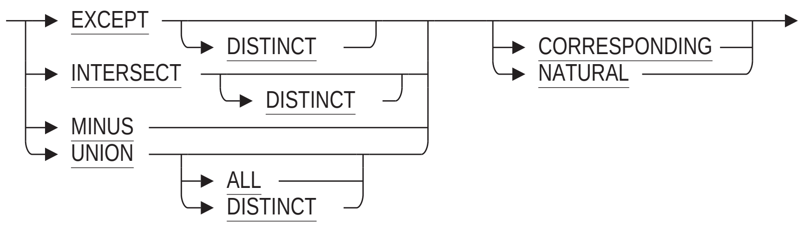
select-expr =



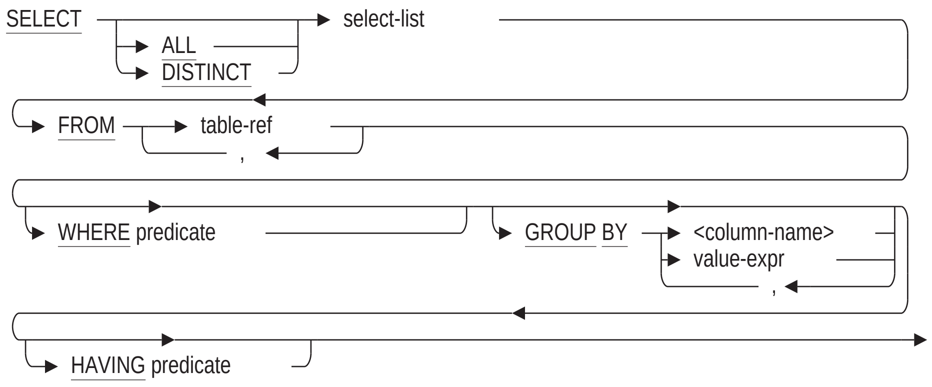
with-clause =



select-merge-clause =



select-clause =



SELECT Statement: General Form

select-list =

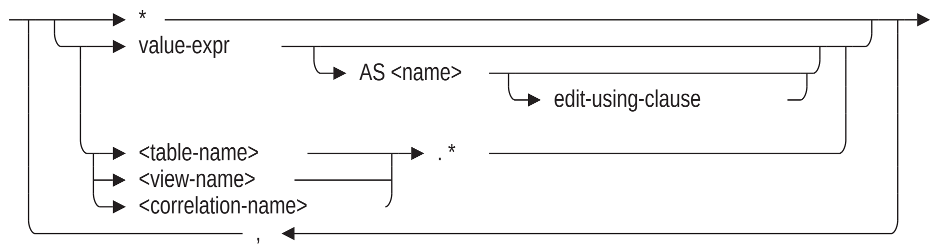
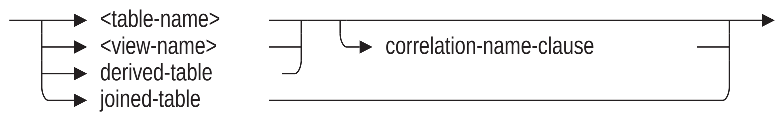
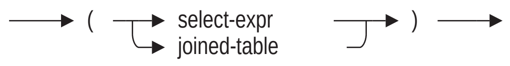


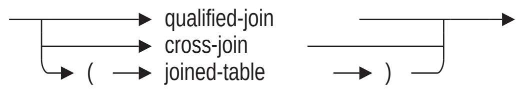
table-ref =



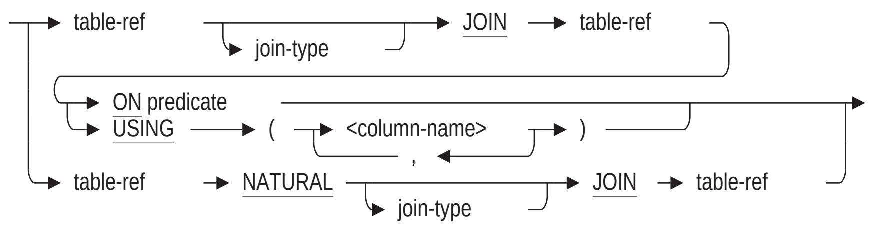
derived-table =



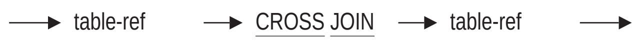
joined-table =



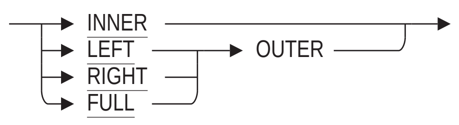
qualified-join =



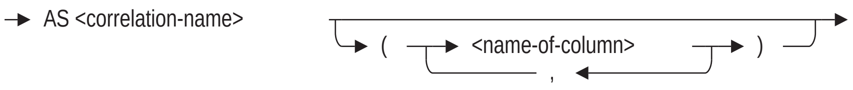
cross-join =



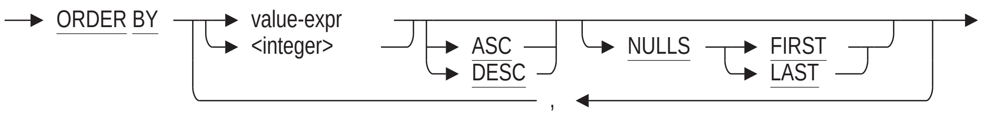
join-type =



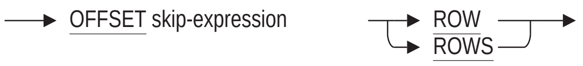
correlation-name-clause =



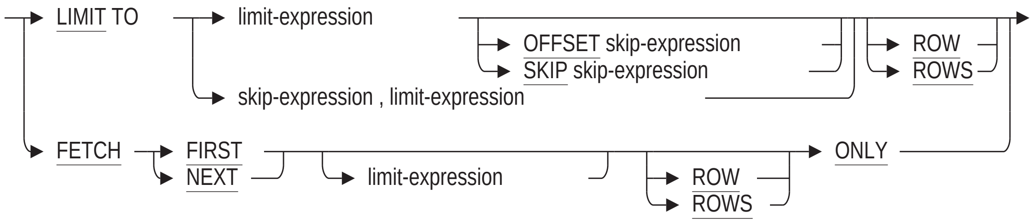
order-by-clause =



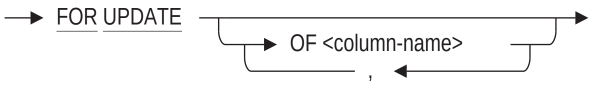
offset-clause =



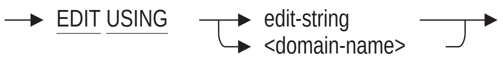
limit-to-clause =



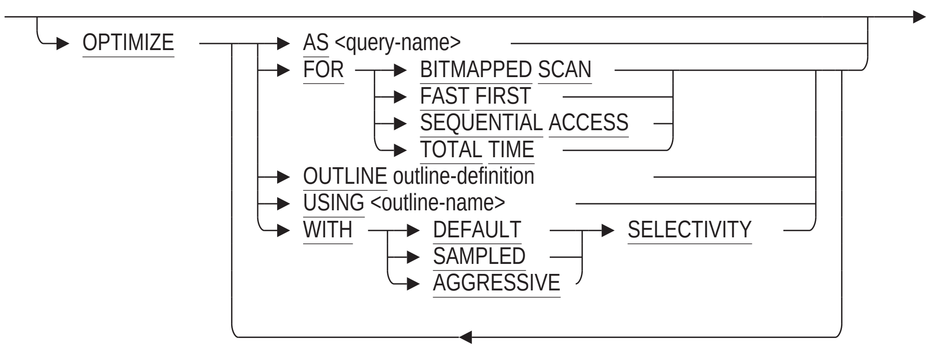
for-update-clause =

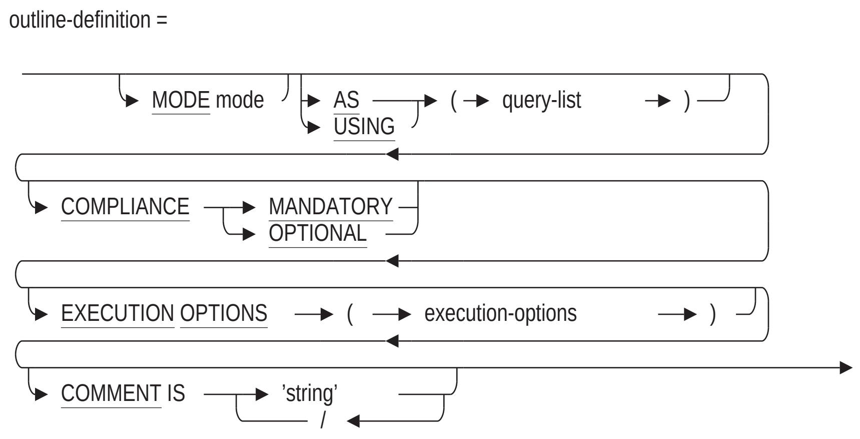


edit-using-clause =



optimize-clause =





Arguments

EDIT USING edit-string
EDIT USING domain-name
Associates an edit string with a value expression. This clause overrides any EDIT STRING defined for the columns or variables in the query. This clause is only permitted for interactive SQL.

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.

The FOR UPDATE OF clause in a SELECT statement provides UPDATE ONLY CURSOR semantics by locking all the rows selected.

OPTIMIZE AS query-name
Assigns a name to the query. You can define the RDMS$DEBUG_FLAGS logical name or use SET FLAGS with the option 'STRATEGY' 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
```

OPTIMIZE FOR

Specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- BITMAPPED SCAN

  Requests the Rdb query optimizer attempt to use BITMAPPED SCAN if there exists multiple supporting indices. This option is not compatible with SEQUENTIAL ACCESS.

- 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 the optimization level is not explicitly set, FAST FIRST is the default.

- SEQUENTIAL ACCESS

  Forces the use of sequential access. This is particularly valuable for tables that use the strict partitioning functionality.

  When the storage map of a table has the attribute PARTITIONING IS NOT UPDATABLE, the mapping of data to a storage area is strictly enforced. This is known as strict partitioning. When queries on such tables use sequential access, the optimizer can eliminate partitions which do not match the WHERE restriction rather than scan every partition.

  The following example shows a query that deletes selected rows from a specific partition. This table also includes several indexes, which may be chosen by the optimizer. Therefore, the OPTIMIZE clause forces sequential access.

  ```
  SQL> delete from PARTS_LOG
  cont> where parts_id between 10000 and 20000
  cont>   and expire_date < :purge_date
  cont> optimize for sequential access;
  ```

  Note that all access performed by such queries will be sequential. Care should be taken that the I/O being used is acceptable by comparing similar queries using index access.

- 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 OUTLINE outline-definition
The OPTIMIZE OUTLINE clause declares a temporary query outline to be used with the select expression.

See the CREATE OUTLINE Statement for more information on defining an outline.

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 WITH
Selects one of three optimization controls: DEFAULT (as used by previous versions of Oracle Rdb), AGGRESSIVE (assumes smaller numbers of rows will be selected), and SAMPLED (which uses literals in the query to perform preliminary estimation on indices).

select-expr
See Section 2.8.1 for a detailed description of select expressions.

Usage Notes

- 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
```

Example 5: Associating an Edit String with a Value Expression

```
SQL> CREATE DOMAIN MONEY INTEGER(2)
cont> EDIT STRING '$$$,$$$,$$9.99';
SQL> --Calculate the average salary for all current jobs.
SQL> SELECT EMPLOYEE_ID,
cont> AVG(SALARY_AMOUNT) AS AVERAGE EDIT USING MONEY,
cont> MAX(SALARY_AMOUNT) AS MAXIMUM EDIT USING MONEY,
cont> MAX(SALARY_START) AS START_DATE EDIT USING 'YYYBDDBMMMBWWW'
cont> FROM SALARY_HISTORY
cont> WHERE SALARY_END IS NULL
cont> GROUP BY EMPLOYEE_ID;
 EMPLOYEE_ID          AVERAGE          MAXIMUM   START_DATE
 00164            $51,712.00       $51,712.00   983 14 Jan Fri
 00165            $11,676.00       $11,676.00   982  1 Jul Thu
 00166            $18,497.00       $18,497.00   982  7 Aug Sat
 00167            $17,510.00       $17,510.00   982 21 Aug Sat
   .
   .
   .
 00435            $84,147.00       $84,147.00   982 12 Mar Fri
 00471            $52,000.00       $52,000.00   982 15 Aug Sun
100 rows selected
```

Example 6: Using the ORDER BY Clause with a Value Expression

```
SQL> SELECT * FROM EMPLOYEES
cont> ORDER BY EXTRACT (YEAR FROM BIRTHDAY),
cont>  TRIM(FIRST_NAME) || TRIM(LAST_NAME);
 00190         O'Sullivan       Rick        G.
   78 Mason Rd.               NULL                        Fremont
      NH      03044         M      12-Jan-1923   1            None
 00231         Clairmont        Rick        NULL
   92 Madiso7 Drive           NULL                        Chocorua
      NH      03817         M      23-Dec-1924   2            None
 00183         Nash             Walter      V.
   197 Lantern Lane           NULL                        Fremont
      NH      03044         M      19-Jan-1925   1            None
 00177         Kinmonth         Louis       NULL
   76 Maple St.               NULL                        Etna
      NH      03750         M       7-Apr-1926   1            None
 00240         Johnson          Bill        R.
   20 South St                NULL                        Milford
      NH      03055         M      13-Apr-1927   2            None
   .
   .
   .
```

Example 7: Using the GROUP BY Clause with a Value Expression

```
SQL> SELECT COUNT (*), EXTRACT (YEAR FROM BIRTHDAY)
cont> FROM EMPLOYEES
cont> GROUP BY EXTRACT (YEAR FROM BIRTHDAY);
           1          1923
           1          1924
           1          1925
           1          1926
           4          1927

           2          1928
           1          1930
           2          1931
   .
   .
   .
```

Example 8: Performing an Outer Join with Oracle Database Style Syntax

```
SQL> SELECT EMPLOYEES.EMPLOYEE_ID, JOB_CODE
cont> FROM EMPLOYEES, CURRENT_JOB
cont> WHERE EMPLOYEES.EMPLOYEE_ID= CURRENT_JOB.EMPLOYEE_ID(+);
 EMPLOYEES.EMPLOYEE_ID   CURRENT_JOB.JOB_CODE
 00164                   DMGR
 00165                   ASCK
 00166                   DMGR
 00167                   APGM
 00168                   DMGR
 00169                   SPGM
 00170                   SCTR
 00171                   PRGM
   .
   .
   .
```

Example 9: The following example shows a query modified with an outline.

```
SQL> set flags 'strategy,detail(2),request_name';
SQL> select last_name, middle_initial, first_name
cont> from employees2
cont> where last_name = 'Toliver' and first_name = 'Alvin'
cont> optimize
cont>     as test3
cont>     outline (
cont>         mode 0
cont>         as (
cont>           query (
cont>             subquery (
cont>               EMPLOYEES2 0  access path index  E3_INDEX
cont>               )
cont>             )
cont>           )
cont>         compliance optional
cont>         execution options (total time)
cont>         );
~Query Name: "TEST3"
~S: Outline "(unnamed)" used
Tables:
  0 = EMPLOYEES2
Leaf#01 BgrOnly 0:EMPLOYEES2 Card=100
  Bool: (0.LAST_NAME = 'Toliver') AND (0.FIRST_NAME = 'Alvin')
  BgrNdx1 E3_INDEX [1:1] Fan=14
    Keys: 0.LAST_NAME = 'Toliver'
 LAST_NAME        MIDDLE_INITIAL   FIRST_NAME
 Toliver          A.               Alvin
1 row selected
SQL>
```

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.

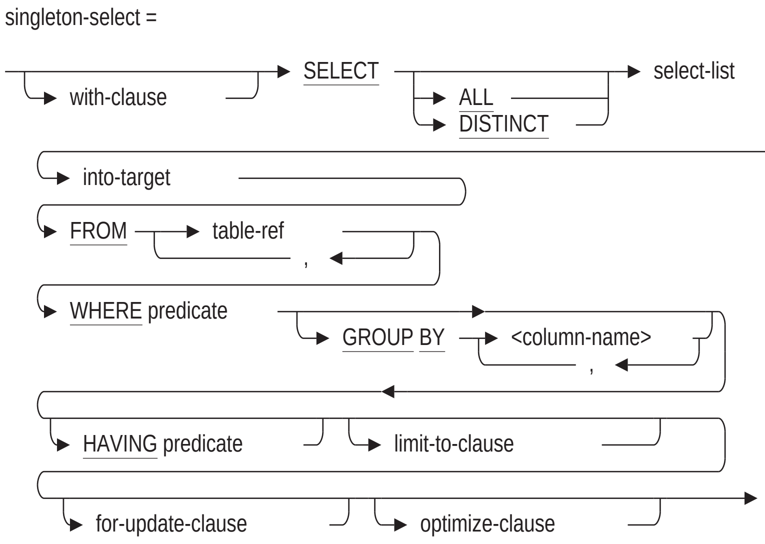
For information on the general form of the SELECT statement, see the SELECT Statement: General Form.

Environment

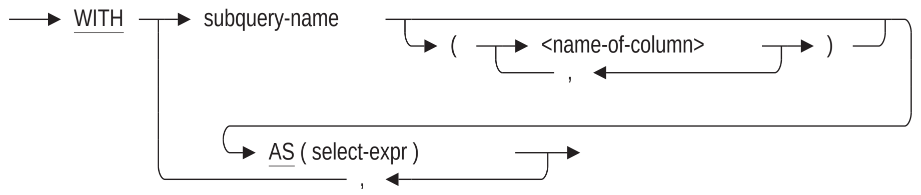
You can use a singleton select statement:

- 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

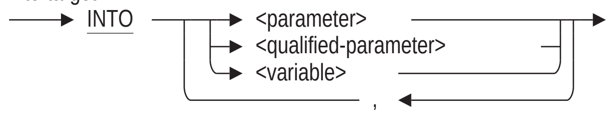
Format



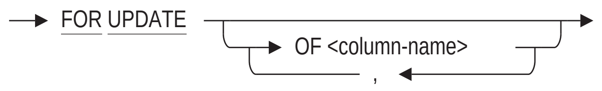
with-clause =



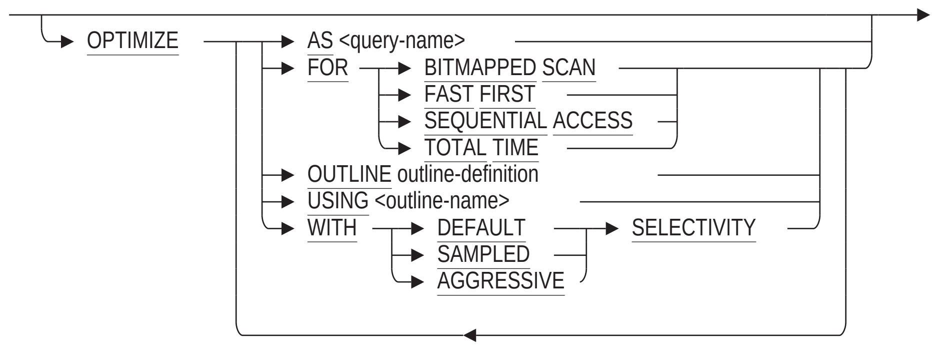
into-target =



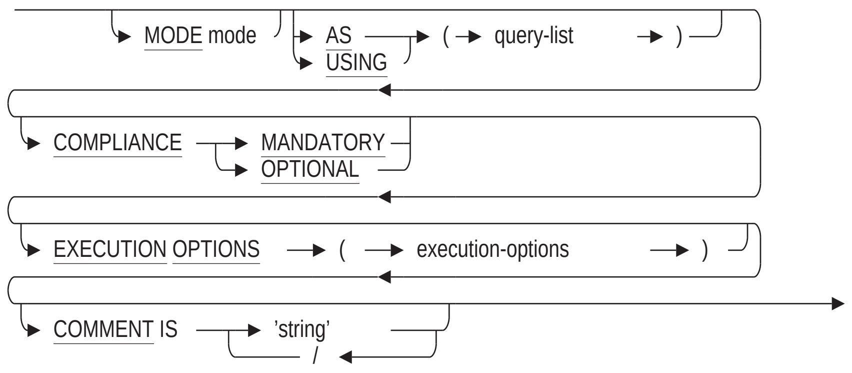
for-update-clause =



optimize-clause =



outline-definition =



Arguments

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.

select-list
For a description of select lists, 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:

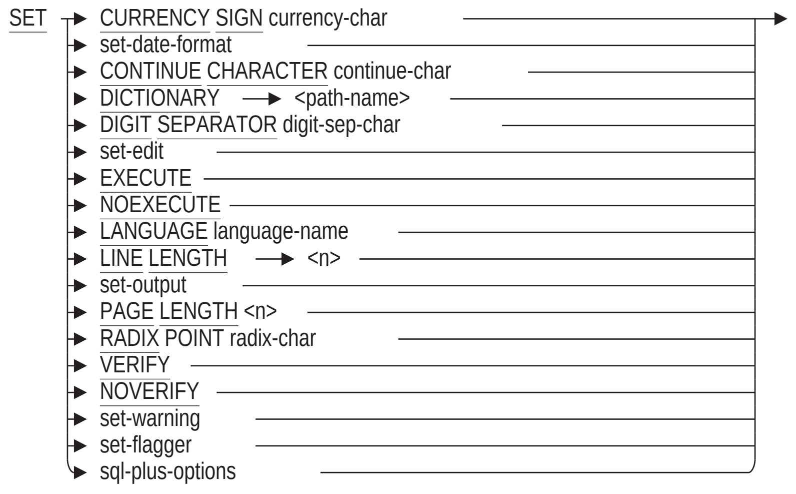
- Currency indicator to be displayed for output
- Display format for date values, time values, or both
- Default path name in the data dictionary
- Digit separator to be displayed for output
- Number of statements to be included in the editing buffer when you type EDIT *
- Language to be used for month abbreviations, and so on, in date and time input and display
- Length of lines to be displayed for output
- Page length for HELP display
- File in which the session is recorded
- 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
- Display of warning messages about nonstandard syntax
- Continue character

Environment

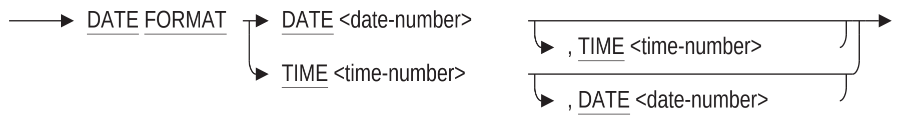
You can use these SET statements in interactive SQL only.

Format

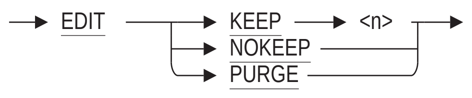
SET Statement



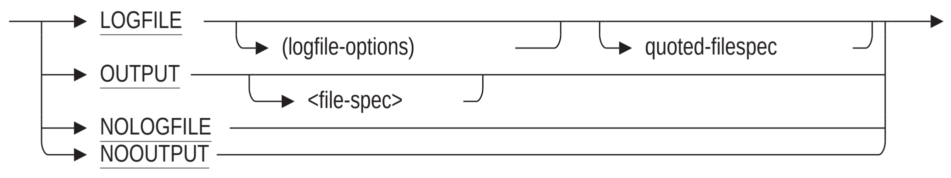
set-date-format=



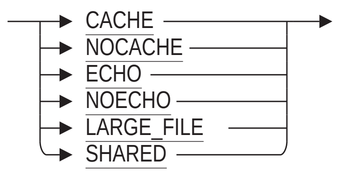
set-edit=



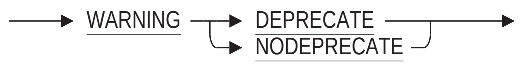
set-output=



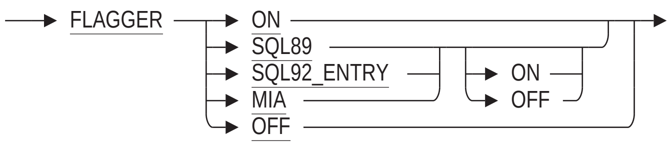
logfile-options =



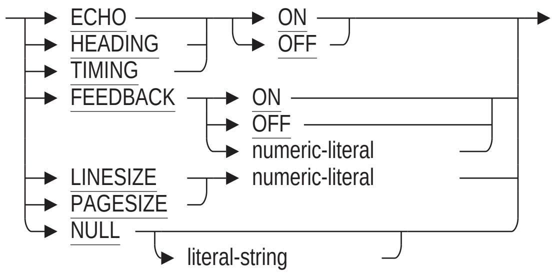
set-warning=



set-flagger =



sql-plus-options =



Arguments

CONTINUE CHARACTER
Defines the continuation character for interactive SQL. By selecting a seldom used character the database administrator can avoid problems with the minus sign to use a continuation character in scripts.

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 ($) or the value specified by the logical name SYS$CURRENCY.

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.

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.

DICTIONARY path-name
Changes your default repository path name to the path name you specify.

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 the value specified by the logical name SYS$DIGIT_SEP.

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.

EXECUTE
NOEXECUTE
NO EXECUTE
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 the SET 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. SQL also allows you to specify NO EXECUTE (as two words); this has the same meaning as NOEXECUTE.

If you do not specify EXECUTE or NOEXECUTE, the default is EXECUTE.

The SET TRANSACTION statement is not executed when SET NO EXECUTE is active. Start or declare a transaction prior to using SET NO EXECUTE.

FLAGGER OFF
Disables all previously set flaggers indicating nonstandard syntax. This is the default.

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.

FEEDBACK { ON | OFF | n }
SET FEEDBACK sets a limit value which turns on feedback only if more than 'n' rows are displayed. SQL data manipulation statements such as SELECT, DELETE, UPDATE, and INSERT will display the number of affected rows.

- SET FEEDBACK 0 is equivalent to SET FEEDBACK ON.
- SET FEEDBACK ON is a synonym for the SET DISPLAY ROW COUNTER statement.

- SET FEEDBACK OFF is a synonym for the SET DISPLAY NO ROW COUNTER statement.

```
SQL> set feedback 2
SQL> select * from work_status;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 0             INACTIVE      RECORD EXPIRED
 1             ACTIVE        FULL TIME
 2             ACTIVE        PART TIME
3 rows selected
SQL> set feedback 4
SQL> select * from work_status;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 0             INACTIVE      RECORD EXPIRED
 1             ACTIVE        FULL TIME
 2             ACTIVE        PART TIME
```

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.

LINE LENGTH n
LINESIZE 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 65535 octets.

You can use the SET LINE LENGTH (or SET LINESIZE) statement to specify an alternate width for output that you are sending to a file or to an alternate output device.

LOGFILE quoted-filespec

This statement allows the executing SQL script to save output to an OpenVMS file. Output and errors from interactive SQL, as well as those statements, will be written to the file-spec specified.

The SET LOGFILE is functionally equivalent to the SET OUTPUT statement. A SET LOGFILE command that does not specify a file is equivalent to SET NOLOGFILE.

Various keywords can be used to control the written output file.

- CACHE

  This is the default. The OpenVMS file caching will be in effect for this file.
- NOCACHE

  This option disables the OpenVMS file caching for this file. Use this to prevent unnecessary caching for a temporary file.
- ECHO

  This is the default. As well as writing the output to the designated file, all commands and errors generated by interactive SQL are also written to SYS$OUTPUT.
- NOECHO

  If the option NOECHO is used, output to SYS$OUTPUT is disabled. All commands and errors generated by interactive SQL are only written to the output file.
- LARGE_FILE

  If the output written to the LOGFILE is lengthy (such as when capturing the output from a query), then this option will use an RMS EXTENT size of 8192. This might improve output performance for very large files.
- SHARED

  The file is created with the shared attribute which will allow other processes to open and read that file while it is being written by SQL.

NOLOGFILE
Closes the current output file specified by a prior SET LOGFILE (or SET OUTPUT command).

NOOUTPUT
Suspends writing to the output file.

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.

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.

If you disable boldface type using the following DCL command, your log file will not contain escape characters:

```
$ SET TERM/NOANSI_CRT
```

PAGE LENGTH n
PAGESIZE n
Sets the size of a page in SQL help.

The following notes apply to the PAGE LENGTH (or PAGESIZE) clause:

- The integer value must be a value between 10 and 32767.
- SET PAGE LENGTH (or SET PAGESIZE) can be used to effectively disable the paging performed by help by setting the length to a high value such as 32000.
- The page length is automatically set upon entry to interactive SQL and is based on the OpenVMS terminal setting for this session.
- The SHOW DISPLAY command can be used to view the currently defined page length.

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 the value specified by the logical name SYS$RADIX_POINT.

sql-plus-options
These statements are provided for use with SQL*Plus scripts that are run against Oracle Rdb.

Table 8–3 Supported SQL*Plus SET statements

SQL*Plus command	Equivalent Oracle Rdb statement
SET ECHO ON	SET VERIFY
SET ECHO OFF	SET NOVERIFY
SET HEADING ON	SET DISPLAY QUERY HEADER
SET HEADING OFF	SET DISPLAY NO QUERY HEADER
SET FEEDBACK ON	SET DISPLAY ROW COUNTER
SET FEEDBACK OFF	SET DISPLAY NO ROW COUNTER
SET NULL	SET DISPLAY DEFAULT NULL STRING
SET NULL 'literal'	SET DISPLAY NULL STRING 'literal'

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.

TIMING { ON|OFF }
The SET TIMING statement enables a single line report of used CPU and Elapsed time for each successful SQL statement or command.

```
SQL> start transaction;
SQL> set timing on;
SQL> select count(*)
cont> from employees
cont>      inner join job_history using (employee_id)
cont>      inner join salary_history using (employee_id)
cont>      inner join departments using (department_code)
cont>      inner join jobs using (job_code)
cont>      left outer join resumes using (employee_id)
cont>      left outer join degrees using (employee_id)
cont>      left outer join colleges using (college_code)
cont>
cont> ;

        3871
1 row selected
Timing: Elapsed:    0 00:00:00.82 Cpu:    0 00:00:00.16
SQL> set timing off;
SQL> commit;
```

VERIFY
Displays indirect command files at your terminal as you run them.

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.

Usage Notes

- 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.
- You cannot use the SET LANGUAGE statement in dynamic SQL; instead, you should use the logical name SYS$LANGUAGE as documented in Table 8–4.
- 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.
- To produce the default currency indicator or digit separator, you must specify an edit string for that column or use the EDIT USING clause on SELECT.
- Table 8–4 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 8–4 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.

- 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.
- The continuation character must be a valid SQL language terminator. These characters are: ’#’, ’(’, ’)’, ’*’, ’+’, ’,’, ’-’, ’.’, ’/’, ’:’, ’;’, ’?’, ’[’, ’\’, ’]’, ’{’, ’ | ’, and ’}’.
- Currently only single octet values are supported by Interactive SQL.
- Use the SHOW CONTINUE CHARACTER to display the current continuation character.

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
  ```

- The SET DICTIONARY statement changes the default repository path name.
- 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

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:

```
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
   .
   .
   .
```

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 receive 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 5: 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> DISCONNECT ALL;
SQL> --
SQL> -- When you specify SET WARNING NODEPRECATE, SQL does not display warning
SQL> -- messages.
SQL> --
SQL> SET WARNING NODEPRECATE;
SQL> DECLARE SCHEMA FILENAME personnel;
SQL> DISCONNECT ALL;
```

Example 6: Setting page length

The following example uses the SET PAGE LENGTH command to change the pagination length of HELP.

```
SQL> set page length 40;
SQL> show display
Output of the query header is enabled
Output of the row counter is enabled
Output using edit strings is enabled
Page length is set to 40 lines
Line length is set to 80 bytes
Display NULL values using "NULL"
```

Example 7: Saving the output from a script

The following example shows the use of SET LOGFILE to save the output from a script without echoing the results.

1. The script being executed.

```
set verify;
start transaction read only;
set logfile (noecho) 'saved_date.log';
select rdb$flags from rdb$database;
set nologfile;
show alias;
rollback;
```

2. The output as seen during the Interactive SQL session.

```
SQL> start transaction read only;
SQL>
SQL> set logfile (noecho) 'saved_date.log';
SQL>
SQL> show alias;
Default alias:
    Oracle Rdb database in file SQL$DATABASE
SQL> rollback;
```

3. The output saved in the log file.

```
SQL>
SQL> select rdb$flags from rdb$database;
   RDB$FLAGS
           0
1 row selected
SQL>
SQL> set nologfile;
```

SET ALIAS Statement

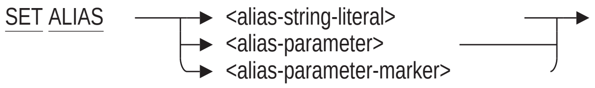
Specifies the default alias for an SQL user session in dynamically prepared and executed or interactive SQL until another SET ALIAS statement is issued. If you do not specify an alias, the default is RDB$DBHANDLE.

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-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.

alias-string-literal
Specifies a character string literal that specifies the default alias. The alias string literal must be enclosed in single quotation marks.

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> ATTACH 'ALIAS corp FILENAME corporate_data';
SQL> --
SQL> -- The default alias, RDB$DBHANDLE, refers to PERSONNEL
SQL> -- to simplify references to CORPORATE_DATA make this
SQL> -- database the default alias
SQL> --
SQL> SET ALIAS 'CORP';
   .
   .
   .
```

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.

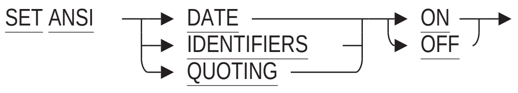
In addition, SQL provides the SET DIALECT statement to let you specify, with one statement, settings for all of these statements. See the SET DIALECT Statement for more information.

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 data type, and the data type of the CURRENT_TIMESTAMP function.

The DATE and CURRENT_TIMESTAMP data types, can be either VMS ADT or ANSI. By default, both data types are interpreted as DATE VMS. The VMS 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 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 Appendix 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> begin
cont> declare :logging_date timestamp;
cont> set :logging_date = current_timestamp;
cont> trace :logging_date;
cont> end;
%SQL-F-UNSDATASS, Unsupported date/time assignment from <Source> to LOGGING_DATE
SQL> SET ANSI DATE ON;
SQL> begin
cont> declare :logging_date timestamp;
cont> set :logging_date = current_timestamp;
cont> trace :logging_date;
cont> end;
```

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);
SQL>
```

SET AUTOMATIC TRANSLATION Statement

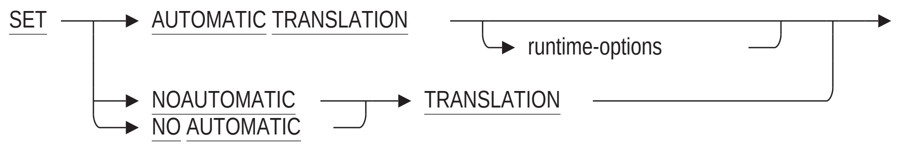
Enables or disables automatic translation to and from the display character set.

Environment

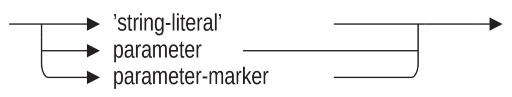
You can use the SET AUTOMATIC TRANSLATION statement:

- 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

'string-literal'
parameter
parameter-marker
Specifies the value of runtime-options, which must be one of the following:

- ON
- OFF

ON enables automatic character set translation and OFF disables it. If no runtime-options are specified, then the default behavior is to enable automatic translation.

Usage Notes

- Enables the automatic translation of character data between client applications and the Oracle Rdb server. This means that column data is translated to the display character set during retrieval, and database object names in queries are converted to the identifier character set during query processing. See SET DISPLAY CHARACTER SET Statement for more information.

- SET AUTOMATIC TRANSLATION will affect all databases in the current environment. If no databases are attached then this setting will be applied as databases are attached.
- The SET NO AUTOMATIC TRANSLATION and SET NOAUTOMATIC TRANSLATION statements may only be used in Interactive SQL. They are equivalent to SET AUTOMATIC TRANSLATION OFF.
- If AUTOMATIC TRANSLATION is enabled then translation is attempted between different versions of the table row. For instance, after an ALTER TABLE command where a new character set is specified for existing data. This is demonstrated in the following example.

```
SQL> create table SAMPLE (description char(20));
SQL> insert into SAMPLE (description) values ('Sample text');
1 row inserted
SQL> select description from SAMPLE;
 DESCRIPTION
 Sample text
1 row selected
SQL> alter table SAMPLE modify (description char(20) character set utf8);
SQL> select description from SAMPLE;
%RDB-E-CONVERT_ERROR, invalid or unsupported data conversion
-RDMS-E-CSETBADASSIGN, incompatible character sets prohibit the requested
assignment
SQL> set automatic translation;
SQL> select description from SAMPLE;
 DESCRIPTION
 Sample text
1 row selected
SQL>
```

  Note that once the restructuring from an old version is created in the current session, it is not undone by disabling AUTOMATIC TRANSLATION.

Examples

Example 1: Using SET AUTOMATIC TRANSLATION command from a SQL Module Language procedure

```
procedure SET_AUTO_TRANS (sqlcode);
    SET AUTOMATIC TRANSLATION ON;
```

```
Or if a parameter is passed:
```

```
procedure SET_AUTO_TRANS
    (sqlcode,
    :on_off char(3)
    );
    SET AUTOMATIC TRANSLATION :on_off;
```

Example 2: Using SET AUTOMATIC TRANSLATION at runtime

```
SQL> declare :auto_trans char(10);
SQL> accept :auto_trans;
Enter value for AUTO_TRANS: off
SQL> set automatic translation :auto_trans;
SQL> show automatic translation;
Automatic translation: OFF
SQL>
```

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.

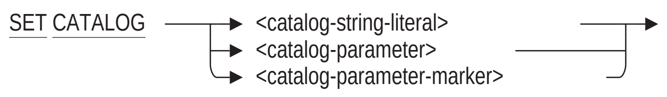
Within one multischema database, tables in different catalogs can be used in a single SQL statement; tables in catalogs in different databases cannot. If you omit the catalog name when you specify an object in a multischema database, SQL uses the default catalog name.

Environment

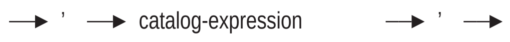
You can use the SET CATALOG statement:

- 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

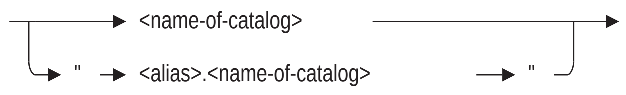
Format



catalog-string-literal =



catalog-expression =



Arguments

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 2.2.3 for more information on catalogs.

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-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.

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 2.2.11. 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

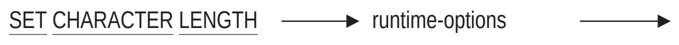
Specifies whether the length of character string parameters, columns, domains, and offsets are interpreted as characters or octets. (An **octet** is a group of 8 bits.)

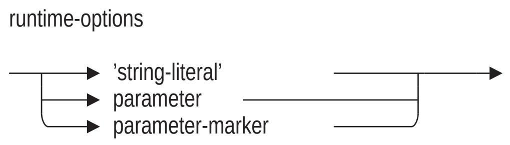
Environment

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





Arguments

'string-literal'
parameter
parameter-marker
Specifies the value of runtime-options, which must be one of the following:

- OCTETS
- CHARACTERS

CHARACTERS specifies the length of character string parameters, columns, domains, and offsets, which are interpreted as characters.

OCTETS specifies the length of character string parameters, columns, domains, and offsets, which are interpreted as octets.

The default is octets.

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> show connection current;
Connection: RDB$DEFAULT_CONNECTION
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
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
Display character set is UNSPECIFIED

Alias RDB$DBHANDLE:
        Identifier character set is DEC_MCS
        Default character set is DEC_MCS
        National character set is DEC_MCS
SQL> /*
***> Create two domains: one uses LATIN9, a single-octet character
***> set, and one uses KANJI a fixed multi-octet character set.
***> */
SQL> create domain LATIN9_DOM char(8) character set ISOLATIN9;
SQL> create domain KANJI_DOM char(5) character set KANJI;
%SQL-F-CHRUNIBAD, Number of octets is not an integral number of characters
SQL> /*
***> Because KANJI is a fixed multi-octet character set, using two
***> octets for each character, you must specify the size as a
***> multiple of two.
***> */
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
LATIN9_DOM                      CHAR(8)
         ISOLATIN9 8 Characters,  8 Octets
SQL>
```

Example 2: Setting the character length to characters

```
SQL> set character length 'characters';
SQL> show connection current;
Connection: RDB$DEFAULT_CONNECTION
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
Dialect: SQLV40
Default character unit: CHARACTERS
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
Display character set is UNSPECIFIED

Alias RDB$DBHANDLE:
        Identifier character set is DEC_MCS
        Default character set is DEC_MCS
        National character set is DEC_MCS
SQL> /*
***> Create two domains: one uses LATIN9, a single-octet character
***> set, and one uses KANJI a fixed multi-octet character set.
***> */
SQL> create domain LATIN9_DOM char(8) character set ISOLATIN9;
SQL> create domain KANJI_DOM char(5) character set KANJI;
SQL> show domains;
User domains in database with filename MIA_CHAR_SET
KANJI_DOM                       CHAR(5)
         KANJI 5 Characters,  10 Octets
LATIN9_DOM                      CHAR(8)
         ISOLATIN9 8 Characters,  8 Octets
SQL>
```

SET COMPOUND TRANSACTIONS Statement

Allows you to control the SQL behavior for starting a default transaction for a compound statement.

By default, if there is no current transaction, SQL starts a transaction before executing a compound statement or stored procedure. However, this might conflict with the actions within the procedure, or it might start a transaction for no reason if the procedure body does not perform any database access. This default is retained for backward compatibility for applications which may expect a transaction to be started for the procedure.

Environment

You can use the SET COMPOUND TRANSACTIONS statement:

- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed

Format

```
SET COMPOUND TRANSACTION  ——▶ int-ext-val
```

Argument

int-ext-value
A string literal or host variable containing the keyword 'INTERNAL' or 'EXTERNAL'. These keywords can be in any case (uppercase, lowercase, or mixed case). If the value is set to EXTERNAL, then SQL starts a transaction before executing the procedure. If the value is set to INTERNAL, then SQL allows the procedure to start a transaction as required by the procedure execution.

Usage Notes

- In the SQL module language or precompiler header, the COMPOUND TRANSACTIONS option can be used to disable or enable starting a transaction for procedures. The keyword INTERNAL or EXTERNAL must be used to enable or disable this feature.

```
MODULE TXN_CONTROL
LANGUAGE BASIC
PARAMETER COLONS
COMPOUND TRANSACTIONS INTERNAL

PROCEDURE S_TXN (SQLCODE);
BEGIN
SET TRANSACTION READ WRITE;
END;

PROCEDURE C_TXN (SQLCODE);
BEGIN
COMMIT;
END;
```

Example

Example 1: Enabling and Disabling Transaction Starting

In interactive or dynamic SQL, the following SET command can be used to disable or enable transactions starting by the SQL interface. The parameter to the SET command is a string literal or host variable containing the keyword 'INTERNAL' or 'EXTERNAL'.

```
SQL> SET COMPOUND TRANSACTIONS 'internal';
SQL> CALL START_TXN_AND_COMMIT ();
SQL> SET COMPOUND TRANSACTIONS 'external';
SQL> CALL UPDATE_EMPLOYEES (...);
```

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.

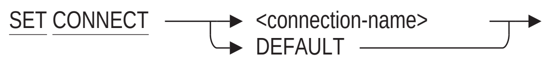
For information about creating and naming connections, see the CONNECT Statement.

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.

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;
->      RDB$DEFAULT_CONNECTION
SQL> --
SQL> -- Start a new connection called CA.
SQL> --
SQL> CONNECT TO 'FILENAME personnel_ca'
cont>     AS 'CA';
SQL> SHOW CONNECTIONS;
        RDB$DEFAULT_CONNECTION
->      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;
->      RDB$DEFAULT_CONNECTION
SQL> --
SQL> -- Even though the database personnel_ca is still attached, CA
SQL> -- is no longer an active connection.
SQL> --
SQL> SET CONNECT 'CA';
%SQL-F-NOSUCHCON, There is not an active connection by that name
SQL> --
SQL> -- The original ATTACH statement comprises the default connection.
SQL> -- The DISCONNECT DEFAULT statement detaches the default connection.
SQL> --
SQL> DISCONNECT DEFAULT;
SQL> SHOW DATABASES;
%SQL-F-ERRATTDEF, Could not use database file specified by SQL$DATABASE
-RDB-E-BAD_DB_FORMAT, SQL$DATABASE does not reference a database known to Rdb
-RMS-E-FNF, file not found
```

SET CONSTRAINTS Statement

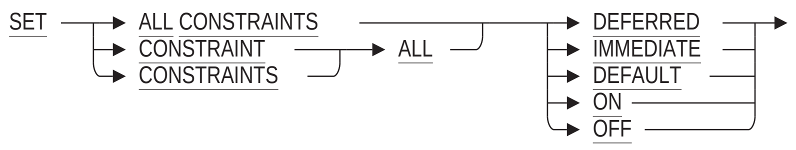
Controls checking for constraints that are evaluated at commit time. (This statement has no effect on constraints that are evaluated at verb time. For verb-time evaluation information, see the SET TRANSACTION Statement.) The SET CONSTRAINTS statement is used to evaluate deferrable constraints at intervals before the transaction is committed.

Environment

You can use the SET 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

DEFAULT
The default constraint mode setting for a session is DEFERRED unless you have used one of the following to specify otherwise:

- SET DEFAULT CONSTRAINT MODE IMMEDIATE statement
- SQLOPTIONS=(CONSTRAINTS=IMMEDIATE) qualifier on the SQL precompiler command line
- CONSTRAINTS=IMMEDIATE qualifier on the SQL module language command line

DEFERRED
OFF
This option causes constraint evaluation to be deferred until commit time, when the transaction completes. OFF is synonymous with DEFERRED.

IMMEDIATE
ON
This option causes constraint evaluation to be executed immediately, when the statement completes. ON is synonymous with IMMEDIATE.

When you issue a SET CONSTRAINTS IMMEDIATE statement, SQL:

- Evaluates all previously deferred constraints (those that would otherwise be evaluated at a COMMIT 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.

Usage Notes

- If a transaction was declared but is not active when the SET CONSTRAINTS statement is executed, SQL starts the declared transaction.
- 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 SET CONSTRAINTS statement only affects when deferrable (commit time) constraints get evaluated. For information about the VERB TIME clause, see the SET TRANSACTION Statement.
- This statement does not affect NOT DEFERRABLE constraints.
- See the *Oracle Rdb Guide to SQL Programming* for information on guidelines for controlling constraint evaluation time.
- The SET CONSTRAINTS ALL ON statement is equivalent to SET CONSTRAINTS ALL IMMEDIATE, and SET CONSTRAINTS ALL OFF is equivalent to SET CONSTRAINTS ALL DEFERRED. The ON and OFF keywords comply with the ANSI/ISO 1989 SQL standard; IMMEDIATE and DEFERRED comply with later ANSI/ISO SQL standards.

Example

Example 1: Using the SET CONSTRAINTS statement in interactive SQL

```
SQL> att 'file mf_personnel_sql';
SQL> set all constraints immediate;
SQL> show constraint;
    Statement constraint evaluation default is DEFERRED (off)
    Statement constraint evaluation is IMMEDIATE (on)
SQL> /*
***> Show the constraints
***> */
SQL> show tables (constraints) job_history;
Information for table 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> set all constraints deferred;
SQL> show constraint;
Statement constraint evaluation default is DEFERRED (off)
Statement constraint evaluation is DEFERRED (off)
SQL>
```

SET Control Statement

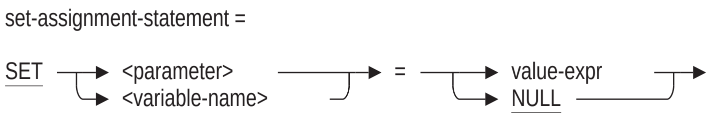
Assigns a value to a target parameter or a variable name.

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



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.
- When assigning a value to a parameter without an indicator parameter 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

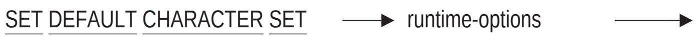
Specifies the default character set for the SQL session.

Environment

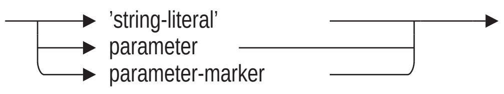
You can use the SET DEFAULT CHARACTER SET statement:

- 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

'string-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 the 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 SQL99, 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.
- 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 SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
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 SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
```

SET DEFAULT CONSTRAINT MODE Statement

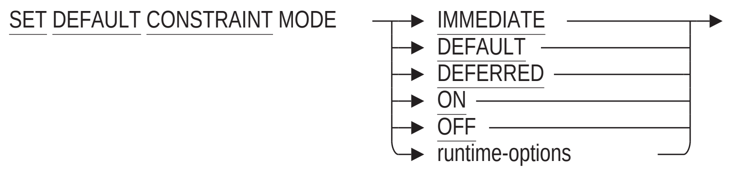
Sets the default constraint setting for statements.

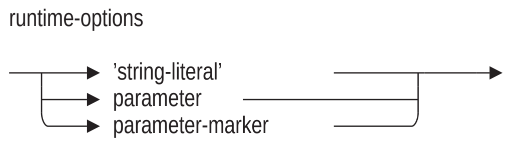
Environment

You can use the SET DEFAULT CONSTRAINT MODE statement:

- In interactive SQL
- In Dynamic SQL as a statement to be dynamically executed

Format





Arguments

DEFAULT
OFF
Requests that during the next transaction, all constraints defined as DEFERRABLE INITIALLY DEFERRED be evaluated as originally specified in the constraint definition. OFF is synonymous with DEFAULT.

DEFERRED
Synonymous with DEFAULT. However, in a future release of Oracle Rdb this keyword will change meaning.

IMMEDIATE
ON
This option requests that during this transaction, all constraints defined as DEFERRABLE INITIALLY DEFERRED be evaluated as though defined as DEFERRABLE INITIALLY IMMEDIATE. ON is synonymous with IMMEDIATE.

′string-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

- Within a transaction the constraint mode can be set temporarily using the SET ALL CONSTRAINTS statement. When a COMMIT or ROLLBACK is executed, the mode will revert to that established by SET DEFAULT CONSTRAINT MODE.
- This statement does not affect the execution of NOT DEFERRABLE constraints.

Examples

Example 1: 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 mf_personnel_sql';
SQL> /*
***> Show settings before starting, set the default mode,
***> then show the settings again.
***> */
SQL> show constraint mode;
    Statement constraint evaluation default is DEFERRED (off)
SQL> set default constraint mode immediate;
SQL> show constraint mode;
    Statement constraint evaluation default is IMMEDIATE (on)
SQL> start transaction;
SQL> set all constraints deferred;
SQL> show constraint mode;
    Statement constraint evaluation default is IMMEDIATE (on)
    Statement constraint evaluation is DEFERRED (off)
SQL> commit;
SQL> show constraint mode;
    Statement constraint evaluation default is IMMEDIATE (on)
SQL>
```

Example 2: Using runtime options

If using runtime-options the passed character value must be one of the keywords: ON, OFF, IMMEDIATE, DEFERRED, or DEFAULT. The following example shows how this can be done in Interactive SQL.

```
SQL> show constraint mode
    Statement constraint evaluation default is DEFERRED (off)
SQL> declare :c_mode char(10) = 'IMMEDIATE';
SQL> set default constraint mode :c_mode;
SQL> show constraint mode
    Statement constraint evaluation default is IMMEDIATE (on)
SQL>
```

SET DEFAULT DATE FORMAT Statement

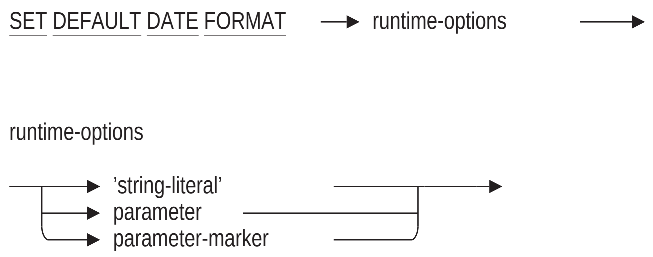
Specifies whether columns with the DATE data type or with the built-in function CURRENT_TIMESTAMP are interpreted as VMS or SQL99 format.

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



Arguments

'string-literal'
parameter
parameter-marker
Specifies the value of runtime-options, which must be one of the following:

- SQL99
- SQL92
- VMS

SQL99 or SQL92
Specifies that the DATE data type and the CURRENT_TIMESTAMP built-in function are interpreted as SQL standard. The SQL standard format DATE contains only the YEAR TO DAY fields, and CURRENT_TIMESTAMP returns a TIMESTAMP data type.

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.

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: Changing the DATE format to SQL99

In the following example, SQL issues an error because, by default, the DATE data type is in OpenVMS DATE format. That is, it contains the fields YEAR through SECOND. The SET DEFAULT DATE FORMAT statement changes the default to ANSI/ISO format so that the CURRENT_DATE and DATE types are compatible.

```
SQL> set default date format 'VMS';
SQL> --
SQL> create domain LOGGING_DATE
cont>     DATE
cont>     default CURRENT_DATE;
%SQL-F-DEFVALINC, You specified a default value for LOGGING_DATE
which is inconsistent with its data type
SQL> --
SQL> set default date format 'SQL99';
SQL> --
SQL> create domain LOGGING_DATE
cont>     DATE
cont>     default CURRENT_DATE;
SQL> show domain LOGGING_DATE;
LOGGING_DATE                    DATE ANSI
 Oracle Rdb default: CURRENT_DATE
```

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 SQL99 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 8–5 shows the settings for each option.

Table 8–5 Dialect Settings

Characteristic	SQL2011[1]	MIA	SQLV40	ORACLE Dialects[2]
Character length	Characters	Characters	Octets	Characters
Quoting rules	Delimited identifier	Delimited identifier	Literal	Delimited identifier
Keywords allowed as identifiers	No	No	Yes	Yes
View update rules	ANSI/ISO SQL rules	ANSI/ISO SQL rules	Oracle Rdb rules	ANSI/ISO SQL rules
Default date format	DATE ANSI	DATE ANSI	DATE VMS	DATE VMS
Default character set	Not changed	KATAKANA	Not changed	Not changed
National character set	Not changed	KANJI	Not changed	Not changed
Identifier character set	Not changed	DEC_KANJI	Not changed	Not changed
Literal character set	Not changed	KATAKANA	Not changed	Not changed
Default evaluation for constraints	Not Deferrable	Deferrable	Deferrable	Not Deferrable

[1]Also applies to SQL92 and SQL99

[2]Applies to ORACLE LEVEL1, ORACLE LEVEL2 and ORACLE LEVEL3

Oracle Corporation recommends that you set the dialect to SQL2011, SQL99 or SQL92 unless you need to maintain compatibility with an earlier dialect. These dialect settings provide enhanced diagnostics and performance in some cases (for example UNIQUE constraints) due to improved SQL Standards.

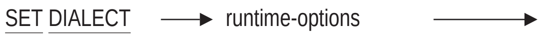
Environment

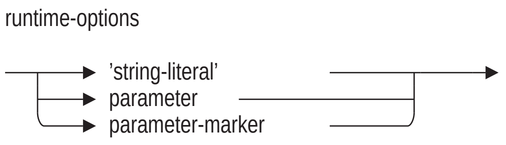
You can use the SET DIALECT statement:

- 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 (but may not be in a compound statement)
- In dynamic SQL as a statement to be dynamically executed

However, the ORACLE dialects can be used only in the interactive SQL and dynamic SQL environments.

Format





Arguments

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 dialects 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.

- The following basic predicate for inequality comparisons is supported:

  !=

  The != basic predicate requires that the ORACLE dialects 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, :1, :2, and so on.
- 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 dialects 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 Oracle Database, a VARCHAR of zero length is considered null. While the Oracle Rdb ORACLE dialects 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 Oracle Database application is to only use views compiled under the ORACLE dialects and to modify tables with VARCHAR columns to remove zero length strings. The following example shows how to remove 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 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 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.

ORACLE LEVEL2

This includes all the behavior describe for ORACLE LEVEL1 plus the following changes:

- The same dialect rules as SQL99 are in effect minus reserved word checking and the DATE ANSI format.
- Concatenate (| |) and the CONCAT function allow for all data types, not just character types (CHAR, and VARCHAR). The numeric, or date/time values are converted to VARCHAR prior to the concatenation.
- Date subtraction results in a floating result. Partial days are now represented by a fraction portion.
- This is not an exhaustive list. Refer to *Oracle Rdb Release Notes* for additional semantic changes for dialect ORACLE LEVEL2.

ORACLE LEVEL3

This includes all the behavior described for ORACLE LEVEL2 plus the following changes:

- The same dialect rules as SQL2011 are in effect.
- The DATE data type is assumed to mean ANSI style DATE which does not include time fields.
- The CURRENT_TIMESTAMP builtin function returns a TIMESTAMP(2) type.

- This is not an exhaustive list. Refer to *Oracle Rdb Release Notes* for additional semantic changes for dialect ORACLE LEVEL3.

′string-literal′
parameter
parameter-marker
Specifies the value of the runtime-options, see **parameter** for the list of keywords.

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
- The constraint evaluation time is DEFERRABLE INITIALLY DEFERRED.

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 SQL format. The SQL (ANSI) 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 2.3.8.2.
- 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.
- The default on constraint evaluation time is set to NOT DEFERRABLE INITIALLY IMMEDIATE.

SQL99

Specifies that the SQL language conforms to SQL:1999 SQL Database Language Standard.

This includes all the behavior described for SQL92 plus the following changes:

- The FOREIGN KEY constraint may list the column names in the REFERENCES list in any order. In other dialects, the column names must be in the same order as the referenced PRIMARY KEY or UNIQUE constraint.

- For CONCAT and | | operators; any date/time or numeric value expressions are automatically CAST as VARCHAR values.
- This is not an exhaustive list. Refer to the *Oracle Rdb Release Notes* for additional semantic changes for dialect SQL99.

SQL2011

This includes all the behavior described for SQL99 plus the following changes:

- PRIMARY KEY or UNIQUE constraints must be evaluated at the same time or sooner than the referencing FOREIGN KEY constraints. That is, a FOREIGN KEY constraint defined as NOT DEFERRABLE may not reference a PRIMARY KEY or UNIQUE constraint defined as DEFERRABLE.
- This is not an exhaustive list. Refer to *Oracle Rdb Release Notes* for additional semantic changes for dialect SQL2011.

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.
- The constraint evaluation time is DEFERRABLE INITIALLY DEFERRED.

The default is SQLV40.

See Table 8–5 for the setting values of the dialect options.

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.

  For all other dialects, 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.
- Other restrictions that apply to the ROWNUM keyword are:
  - Can be used only with the ORACLE dialects. 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.

Examples

Example 1: Setting the characteristics to SQL92

```
SQL> ATTACH 'ALIAS MIA1 FILENAME MIA_CHAR_SET DISPLAY CHARACTER SET SHIFT_JIS';
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 SMITH
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 SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is KATAKANA
        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 SMITH
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
Display character set is SHIFT_JIS
```

```
Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is KATAKANA
        National character set is KANJI
SQL> --
SQL> -- Change the environment from MIA to SQL99.  Notice that the
SQL> -- session characters DO NOT change from the MIA settings.
SQL> --
SQL> SET DIALECT 'SQL99';
SQL> SHOW CONNECTIONS TEST;
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
Dialect: SQL99
Default character unit: CHARACTERS
Keyword Rules: SQL99
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
Display character set is SHIFT_JIS

Alias MIA1:
        Identifier character set is DEC_KANJI
        Default character set is KATAKANA
        National character set is KANJI
```

Example 2: Saving and restoring dialect in interactive SQL

This example shows the use of declared variables in interactive SQL to save (using GET ENVIRONMENT) and restore the dialect during execution of a script that requires an alternate dialect. This example simply displays the dialect using the SHOW CONNECTION statement.

```
SQL> set dialect 'sql99';
SQL> -- save current dialect
SQL> declare :dialect char(40);
SQL> get environment (session) :dialect = DIALECT;
SQL> print :dialect;
 DIALECT
 SQL99
SQL> set dialect 'oracle level2';
SQL> show connection rdb$default_connection;
Connection: RDB$DEFAULT_CONNECTION
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
Dialect: SQL99 (ORACLE LEVEL2)
Default character unit: CHARACTERS
Keyword Rules: SQL99
View Rules: ANSI/ISO
Default DATE type: DATE VMS
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 DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
Display character set is UNSPECIFIED
SQL>
SQL> -- restore previous dialect
SQL> set dialect :dialect;
SQL> show connection rdb$default_connection;
Connection: RDB$DEFAULT_CONNECTION
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
Dialect: SQL99
Default character unit: CHARACTERS
Keyword Rules: SQL99
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 DEC_MCS
National character set is DEC_MCS
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
Display character set is UNSPECIFIED
SQL>
```

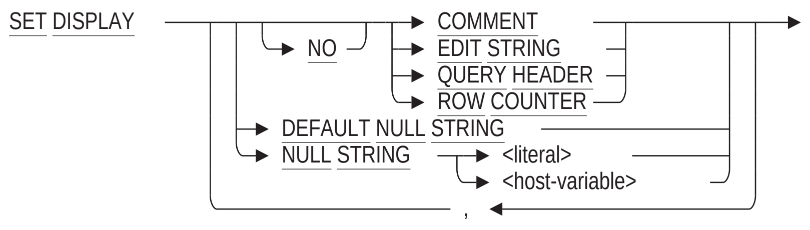
SET DISPLAY Statement

Controls the output of header information. Use the SHOW DISPLAY statement to view the current settings.

Environment

You can use the SET DISPLAY statement in interactive SQL only.

Format



Arguments

COMMENT
NOCOMMENT
Disables or enables the display of comment text by other SHOW commands (for example, SHOW TABLE).

DEFAULT NULL STRING
Reverts to using the text 'NULL'.

EDIT STRING
NO EDIT STRING
Enables the usage of column edit strings to format values for the SELECT statement. Use NO EDIT STRING to disable the use of the column edit strings.

NULL STRING
Changes the way NULL values are displayed by interactive SQL.

QUERY HEADER
NO QUERY HEADER
Enables the printed header generated by the SELECT, CALL, FETCH, and PRINT statements. Use NO QUERY HEADER to disable this header.

ROW COUNTER
NO ROW COUNTER
Enables the total count reported by SELECT, DELETE, INSERT, and UPDATE statements. Use NO ROW COUNTER to disable the trailing count message.

Usage Notes

- The width of the displayed column is calculated using the maximum of the length of the column name, the length of the QUERY HEADER, the length of the NULL string and the size of the formatted data.
- The statement SET DISPLAY DEFAULT NULL STRING is equivalent to SET DISPLAY NULL STRING 'NULL'.
- SET DISPLAY NULL STRING accepts a string literal, or a declared local variable.
- The GET ENVIRONMENT statement includes the NULL_STRING keyword that can be used to save the currently defined text.
- The defaults are to use edit strings, display the query header, and report a row count message. More than one option can be specified, separated by commas. However, you cannot specify both the option and its negated form in one statement, as demonstrated in the following example:

  ```
  SQL> SET DISPLAY QUERY HEADER, NO QUERY HEADER
  %SQL-F-MULTSPECATR, Multiple specified attribute.
  "QUERY HEADER" was specified more than once
  ```
- The following SET statements, provided for compatibility with SQL*Plus, are equivalent to SET DISPLAY clauses:
  - SET HEADING ON is a synonym for the SQL SET DISPLAY QUERY HEADER statement. SQL output statements such as SELECT, PRINT, and FETCH will display the name of the column, variable or its query header.
  - SET HEADING OFF is a synonym for the SQL SET NO DISPLAY QUERY HEADER statement. SQL output statements such as SELECT, PRINT, and FETCH will no longer display the query header.
  - SET FEEDBACK ON is a synonym for the SQL SET DISPLAY NO ROW COUNTER statement. SQL data manipulation statements such as SELECT, DELETE, UPDATE, and INSERT will display the number of affected rows.
  - SET FEEDBACK OFF is a synonym for the SQL SET DISPLAY ROW COUNTER statement. SQL data manipulation statements no longer display the count of affected rows.
  - SET NULL is a synonym for SET DISPLAY NULL STRING ' ', and SET NULL 'literal' is equivalent to SET DISPLAY NULL 'literal'.

Example

Example 1: Using the SET DISPLAY Statement

The following example shows the effect of the SET DISPLAY statement. It uses the SHOW DISPLAY command to report the current settings.

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL>
SQL> CREATE DOMAIN money INTEGER(2) EDIT STRING '$$$,$$9.99';
SQL> CREATE TABLE temp_emp (id INTEGER, sal money);
SQL>
SQL> SELECT * FROM work_status;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 0             INACTIVE      RECORD EXPIRED
 1             ACTIVE        FULL TIME
 2             ACTIVE        PART TIME
3 rows selected
SQL>
SQL> SET DISPLAY NO ROW COUNTER;
SQL> SHOW DISPLAY
Output of the query header is enabled
Output of the row counter is disabled
Output using edit strings is enabled
Page length is set to 24 lines
Line length is set to 132 bytes
Display NULL values using "NULL"
SQL> SELECT * FROM work_status;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 0             INACTIVE      RECORD EXPIRED
 1             ACTIVE        FULL TIME
 2             ACTIVE        PART TIME
SQL> INSERT INTO temp_emp (id) VALUES (0);
SQL> INSERT INTO temp_emp (id, sal)
cont> SELECT employee_id, MAX(salary_amount)
cont>   FROM salary_history GROUP BY employee_id;
SQL> UPDATE temp_emp SET id = NULL WHERE id <= 0;
SQL> DELETE FROM temp_emp WHERE id IS NULL;
SQL>
SQL> SET DISPLAY ROW COUNTER;
SQL> SHOW DISPLAY
Output of the query header is enabled
Output of the row counter is enabled
Output using edit strings is enabled
Page length is set to 24 lines
Line length is set to 132 bytes
Display NULL values using "NULL"
SQL>
SQL> SELECT * FROM work_status;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 0             INACTIVE      RECORD EXPIRED
 1             ACTIVE        FULL TIME
 2             ACTIVE        PART TIME
3 rows selected
SQL>
SQL> SET DISPLAY NO QUERY HEADER;
SQL> SHOW DISPLAY
Output of the query header is disabled
Output of the row counter is enabled
Output using edit strings is enabled
Page length is set to 24 lines
Line length is set to 132 bytes
Display NULL values using "NULL"
SQL>
SQL> DECLARE :res INTEGER;
SQL>
SQL> -- This omits the query header for the SELECT statement
SQL> SELECT * FROM work_status;
 0             INACTIVE      RECORD EXPIRED
 1             ACTIVE        FULL TIME
 2             ACTIVE        PART TIME
3 rows selected
```

```
SQL>
SQL> -- This omits the query header for the PRINT statement
SQL> PRINT :res;
           0
SQL> PRINT 'This is a print line';
 This is a print line
SQL>
SQL> CREATE MODULE call_sample
cont>     LANGUAGE SQL
cont>     PROCEDURE add_one (IN :a INTEGER, OUT :b INTEGER);
cont>     SET :b = :a + 1;
cont> END MODULE;
SQL> -- This omits the query header for the OUT/INOUT parameters for CALL
SQL> CALL add_one (100, :res);
         101
SQL>
SQL> DECLARE c CURSOR FOR SELECT * FROM work_status;
SQL> OPEN c;
SQL> -- This omits the query headers for the variables fetched
SQL> FETCH c;
 0             INACTIVE      RECORD EXPIRED
SQL> SET DISPLAY QUERY HEADER;
SQL> SHOW DISPLAY
Output of the query header is enabled
Output of the row counter is enabled
Output using edit strings is enabled
Page length is set to 24 lines
Line length is set to 132 bytes
Display NULL values using "NULL"
SQL> -- This outputs the query headers for the variables fetched
SQL> FETCH c;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 1             ACTIVE        FULL TIME
SQL> CLOSE c;
SQL>
SQL> TRUNCATE TABLE temp_emp;
SQL> INSERT INTO temp_emp (id, sal)
cont>     SELECT employee_id, AVG(salary_amount)
cont> FROM salary_history
cont> WHERE salary_end IS NULL
cont> GROUP BY employee_id;
100 rows inserted
SQL>
SQL> SELECT * FROM temp_emp ORDER BY id LIMIT TO 3 ROWS;
          ID          SAL
         164   $51,712.00
         165   $11,676.00
         166   $18,497.00
3 rows selected
SQL>
SQL> SET DISPLAY NO EDIT STRING;
SQL> SHOW DISPLAY
Output of the query header is enabled
Output of the row counter is enabled
Output using edit strings is disabled
Page length is set to 24 lines
Line length is set to 132 bytes
Display NULL values using "NULL"
SQL>
SQL> SELECT * FROM temp_emp ORDER BY id LIMIT TO 3 ROWS;
         164     51712.00
         165     11676.00
         166     18497.00
3 rows selected
SQL>
SQL> SET DISPLAY EDIT STRING;
SQL> SHOW DISPLAY
Output of the query header is enabled
Output of the row counter is enabled
Output using edit strings is enabled
Page length is set to 24 lines
Line length is set to 132 bytes
Display NULL values using "NULL"
SQL>
SQL> SELECT * FROM temp_emp ORDER BY id LIMIT TO 3 ROWS;
          ID         SAL
         164   $51,712.00
         165   $11,676.00
         166   $18,497.00
3 rows selected
```

Example 2: Replacing the NULL values with text to make the output easier to read

```
SQL> select job_start, job_end,
cont>             (select department_name
cont>              from departments d
cont>              where d.department_code = jh.department_code)
cont> from job_history jh
cont> where employee_id = '00164';
 JOB_START     JOB_END
 21-Sep-1981   NULL          Board Manufacturing North
  5-Jul-1980   20-Sep-1981   Cabinet & Frame Manufacturing
2 rows selected
SQL> set display null string '(still employeed)'
SQL> select job_start, job_end,
cont>             (select department_name
cont>              from departments d
cont>              where d.department_code = jh.department_code)
cont> from job_history jh
cont> where employee_id = '00164';
 JOB_START           JOB_END
 21-Sep-1981         (still employeed)   Board Manufacturing North
  5-Jul-1980         20-Sep-1981         Cabinet & Frame Manufacturing
2 rows selected
```

Example 3: Disabling the comment display to make the output of SHOW easier to read

```
SQL> show domain id_dom
ID_DOM                          CHAR(5)
 Comment:       standard definition of employee id
SQL> set display no comment;
SQL> show domain id_dom
ID_DOM                          CHAR(5)
SQL>
```

Example 4: Save the current NULL string using GET ENVIRONMENT and restore after executing a query.

```
SQL> declare :ns varchar(100);
SQL> get environment (session) :ns = NULL_STRING;
SQL> set null;
SQL> select job_start, job_end,
cont>             (select department_name
cont>              from departments d
cont>              where d.department_code = jh.department_code)
cont> from job_history jh
cont> where employee_id = '00164';
 JOB_START     JOB_END
 21-Sep-1981                Board Manufacturing North
  5-Jul-1980   20-Sep-1981   Cabinet & Frame Manufacturing
2 rows selected
SQL> set display null string :ns;
SQL> select job_start, job_end,
cont>             (select department_name
cont>              from departments d
cont>              where d.department_code = jh.department_code)
cont> from job_history jh
cont> where employee_id = '00164';
 JOB_START     JOB_END
 21-Sep-1981   NULL          Board Manufacturing North
  5-Jul-1980   20-Sep-1981   Cabinet & Frame Manufacturing
2 rows selected
```

SET DISPLAY CHARACTER SET Statement

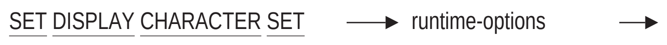
Specifies the display character set.

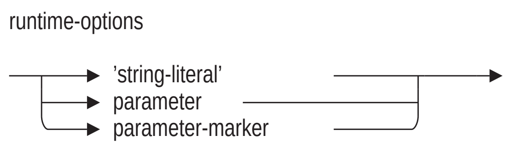
Environment

You can use the SET DISPLAY 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

'string-literal'
parameter
parameter-marker
Specifies the display character set used for the automatic translation of text values before the values are returned to the user application. See Table 2-1 for a list of allowable character sets and option values.

Usage Notes

- The SET DISPLAY CHARACTER SET statement provides a mechanism for specifying the default display character set to be used implicitly by subsequent attach statements if automatic translation has not been disabled. For example the following statements are equivalent:

  ```
  SQL> SET DISPLAY CHARACTER SET 'SHIFT_JIS';
  SQL> ATTACH 'FILENAME MF_PERSONNEL';
  SQL> --
  SQL> -- is equivalent to:
  SQL> --
  SQL> ATTACH 'FILENAME MF_PERSONNEL DISPLAY CHARACTER SET SHIFT_JIS';
  ```

  Both sets of statements will cause the Oracle Rdb server to automatically translate any text information returned to SQL from that database attach session to the SHIFT_JIS character set.

- The display character set is used in conjunction with AUTOMATIC TRANSLATION. If automatic translation is enabled then Oracle Rdb will attempt to translate character data to and from the specified display character set during retrieval and query of the database. See the SET AUTOMATIC TRANSLATION statement.

  The most common use for this feature is for those client applications not running on OpenVMS. For example, the stored data might be in DEC_KANJI and display is required on a Windows client using the SHIFT_JIS character set.
- SET DISPLAY CHARACTER SET changes the identifier and literal character sets, in addition to the display character set. This allows, for instance, applications to query the database passing in literals and table names that are encoded in the SHIFT_JIS character set. Oracle Rdb will translate these names to the appropriate character set based on the target database attributes.
- Use the SHOW CHARACTER SETS statement to see the current display character set in an interactive session.

  ```
  SQL> set display character set 'SHIFT_JIS';
  SQL> show character sets;
  Default character set is DEC_KANJI
  National character set is DEC_MCS
  Identifier character set is SHIFT_JIS
  Literal character set is SHIFT_JIS
  Display character set is SHIFT_JIS
  ```
- The default is the UNSPECIFIED character set which indicates to Oracle Rdb that no translation will be attempted.

Examples

Example 1: Setting the display 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
Display character set is UNSPECIFIED
SQL> set display character set 'SHIFT_JIS';
SQL> show character sets;
Default character set is DEC_MCS
National character set is DEC_MCS
Identifier character set is SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
```

SET FLAGS Statement

Allows enabling and disabling of database system debug flags for the current session.

The literal or host variable passed to this command can contain a list of keywords, or negated keywords, separated by commas. Spaces are ignored. The keywords may be abbreviated to an unambiguous length.

Note

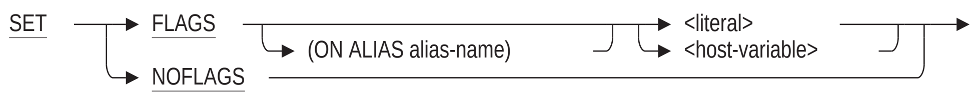
Oracle Corporation reserves the right to add new keywords to the SET FLAGS statement in any release or update to Oracle Rdb, which may change this unambiguous length. Therefore, it is recommended that the full keyword be used in applications.

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.

The Usage Notes show the available keywords that can be specified.

In addition, the keywords (and negated keywords) listed can be specified as the equivalence string for the RDMS$SET_FLAGS logical name.

NOFLAGS
The SET NOFLAGS statement disables all currently enabled flags. It is equivalent to SET FLAGS 'NONE'. NOFLAGS is only permitted in Interactive SQL.

ON ALIAS alias-name
Using the ON ALIAS clause allows the database administrator to set flags on just one database alias instead of using all currently attached databases. Use the name of an alias as declared by the ATTACH or CONNECT statement or, if none was specified, use the default alias name RDB$DBHANDLE.

Usage Notes

- The specified flag is processed by each database to which you are currently attached.
- The SET FLAGS statement overrides the RDMS$DEBUG_FLAGS logical name or the RDMS$SET_FLAGS logical name at the command level.
- 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 does not persist beyond a database attach.
- The RDMS$SET_FLAGS logical name is processed during the attach operation. An exception is raised if an error is found in the equivalence string, and the attach to the database fails. The SQL SHOW FLAGS statement will display settings made with the RDMS$SET_FLAGS and RDMS$DEBUG_FLAGS logical names. Settings made with the RDMS$DEBUG_FLAGS logical name are superseded by keywords specified by RDMS$SET_FLAGS.

  When a generated outline is added to the database it will only be used when the mode is set, either by the SET FLAGS statement or by using the logical name RDMS$BIND_OUTLINE_MODE.
- To set the AUTO_OVERRIDE keyword, you must have the DBADM (administrator) privilege on the database. The DBADM privilege can be granted explicitly or can be inherited from the OpenVMS system privileges.

  If you do not have the required privilege, then the SET FLAG statement fails and returns the NO_PRIV error.
- The AUTO_OVERRIDE flag can be used to allow updates to selected AUTOMATIC columns during INSERT so that rows could be reloaded, or during UPDATE to adjust incorrectly stored values.
  - For the INSERT statement, 'AUTO_OVERRIDE' allows assignment to any AUTOMATIC column, and any AUTOMATIC INSERT column omitted from the column list will be evaluated normally.
  - For the UPDATE statement, 'AUTO_OVERRIDE' allows direct assignment of values to any AUTOMATIC column. No AUTOMATIC columns are evaluated.
- The EXECUTION keyword can be followed immediately by a numeric value in parentheses. This represents the number of lines to display before 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.

- Use VALIDATE_ROUTINE when routines, query outlines, and triggers become invalid due to the following actions:
  - When a table is dropped using the CASCADE option, any procedure or function that references the table is marked invalid.
  - When a table is dropped (using either the CASCADE or RESTRICT options) any query outline that references the table is marked as invalid.
  - When a module is dropped using the CASCADE option, any procedure, function, or query outline that references the module is marked invalid. A query outline references a module when it uses a temporary table declared at the module level.
  - When a routine is dropped using CASCADE, any trigger or routine that references that routine is marked invalid.
  - When an index is dropped, or altered to have MAINTENANCE IS DISABLED, any query outline that references the index is marked as invalid.
- The DATABASE_PARAMETERS keyword generates output only during ATTACH to the database which happens prior to the SET FLAGS statement executing.

  This option is therefore only useful when used with the RDMS$SET_FLAGS logical name.

  ```
  $ define RDMS$SET_FLAGS "database_parameters"
  $ sql$
  SQL> Attach 'File db$:scratch';
   ATTACH #1, Database DISK:[DOCS.V71]SCRATCH.RDB;1
  ~P Database Parameter Buffer (version=2, len=79)
  0000 (00000) RDB$K_DPB_VERSION2
  0001 (00001) RDB$K_FACILITY_ALL
  0002 (00002) RDB$K_DPB2_IMAGE_NAME "NODE::DISK:[DIR]SQL$70.EXE;1"
  0040 (00064) RDB$K_FACILITY_ALL
  0041 (00065) RDB$K_DPB2_DBKEY_SCOPE (Transaction)
  0045 (00069) RDB$K_FACILITY_ALL
  0046 (00070) RDB$K_DPB2_REQUEST_SCOPE (Attach)
  004A (00074) RDB$K_FACILITY_RDB_VMS
  004B (00075) RDB$K_DPB2_CDD_MAINTAINED (No)
   RDMS$BIND_WORK_FILE = "DISK:[DIR]RDMSTTBL$UEOU3LQ0RV2.TMP;" (Visible = 0)
  SQL> Exit
   DETACH #1
  ```
- DDAL (Distributed Data Access Layer) is a generic term applied to the optional Replication Option for Rdb software, which was previously known as DEC Data Distributor.

  This special flag, supported for users of Replication Option for Rdb, allows a knowledgable database administrator to override the normal prohibition that system indices cannot be created or dropped. Several Replication Option for Rdb indices can then be managed to improve performance in specific customer environments. This includes dropping an index and recreating it with a different PAGE SIZE, changing the type from SORTED to SORTED RANKED, or enabling COMPRESSION.

  It also emits a trace message showing that such actions have been performed. The index RDB$TRAN_RELS_REL_NAME_NDX can be managed, and indices can be defined and managed on the table RDB$CHANGES.

- When you use the INDEX_COLUMN_GROUP keyword, applications can make better use of the index column group information specified in indexes. When you do not use this keyword, the Oracle Rdb optimizer may estimate much higher cardinalities for the chosen solution if the selection predicate specifies only some of the leading segments on a multisegment index. This happens, for instance, if you specify an equality on the first segment of a two-segment index.

  This slight overestimation is not a significant problem on relatively small tables but becomes a more significant problem when the select operation involves a sort (in particular, the OpenVMS SORT facility) where the sort buffer is preallocated based on its estimated cardinality of the solution.
- There is no debug flags equivalent for the MODE(n) or NOMODE keywords. Instead, you can use the RDMS$BIND_OUTLINE_MODE logical name.
- You might use the SEQ_CACHE keyword when you are loading many rows with the RMU Load command. This command is most efficient when all of the sequence values are allocated in large batches. For example:

  ```
  $ DEFINE RDMS$SET_FLAGS   "SEQ_CACHE(10000)"
  $ RMU/LOAD/COMMIT_EVERY=50000 DATABASE TABLE FILE
  ```

  In this example, it is assumed that an AUTOMATIC column is defined such that SEQUENCE.NEXTVAL is executed.
- All indices which are created for constraints are of type SORTED. If the database SYSTEM INDEX default is SORTED RANKED then this same default is used by the AUTO_INDEX option.
- Use the INDEX_STATS option with AUTO_INDEX to see a description of the indices which are created.

  If a suitable index already exists then it will be used in preference to creating a new index.

  All indices are created in the DEFAULT storage area, there is no facility to add storage maps for these indices during their creation.

  The index is given the same name as the constraint for which it was created. When the constraint is dropped the index will remain and must be dropped manually. It is possible that the index is used by multiple constraints.
- The SELECTIVITY flag affects user SELECT, UPDATE and DELETE statements provided that those statements do not explicitly or implicitly specify an OPTIMIZE WITH SELECTIVITY clause.
- The TRACE statement 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 for the first time.
- When using interactive or dynamic SQL both WATCH_CALL and WATCH_OPEN will generate trace lines for the queries performed by the SQL runtime system against the Rdb system tables. There is no mechanism to disable the trace of such information.
- The WATCH_CALL and WATCH_OPEN flags cause queries and routines to be modified to output this information. This might add some extra CPU overhead to the application while this information is collected. Even when the flags are disabled there exists some overhead that is not eliminated until the module or query is released, usually at DISCONNECT time.

- You cannot provide an outline name for a query in many situations, such as when you use third party software. In these situations, Oracle Rdb tries to locate an outline with a matching identifier. Because the optimizer generates an identifier as a hashed value that depends on the query structure, small changes in the query, such as different literal values, change the generated identifier.

  You can use the ALTERNATE_OUTLINE_ID(LITERALS) keyword (abbreviated as ALT(LIT)) to control the alternate outline identifiers. Set this keyword by using either the SET FLAGS statement or the RDMS$SET_FLAGS logical name. If this keyword is set, the optimizer discards literal values when producing the identifiers.

```
SQL> set flags 'alt(LIT), outline';
SQL> select * from employees where employee_id = '1';
-- Rdb Generated Outline : 19-SEP-2001 13:52
create outline QO_847AD7287E247D37_00000000
id '847AD7287E247D37E8E4CC8221FFC12E'
mode 0
as (
  query (
-- For loop
    subquery (
      EMPLOYEES 0     access path index       EMP_EMPLOYEE_ID
      )
    )
  )
compliance optional     ;
0 rows selected
SQL> select * from employees where employee_id = '9999';
-- Rdb Generated Outline : 19-SEP-2001 13:52
create outline QO_847AD7287E247D37_00000000
id '847AD7287E247D37E8E4CC8221FFC12E'
mode 0
as (
  query (
-- For loop
    subquery (
      EMPLOYEES 0     access path index       EMP_EMPLOYEE_ID
      )
    )
  )
compliance optional     ;
0 rows selected
```

  You can store this more generic outline to use in any similar query where only the literal values differ, for example:

```
SQL> set flags 'alt(lit)';
SQL> create outline o1 from (select * from employees where employee_id = '1');
SQL> set flags 'strat';
SQL> select * from employees where employee_id = '1';
~S: Outline "O1" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> select * from employees where employee_id = 'AAAAAA';
~S: Outline "O1" used
Conjunct        Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
```

Any outline stored for a query without the ALTERNATE_OUTLINE_ID flag being set will be created using the full query as in previous versions and will take precedence over any generic outline, as seen in the following example.

```
SQL> set noflags;
SQL> create outline o1 from (select * from employees where employee_id = '1');
SQL> set flags 'strat';
SQL> select * from employees where employee_id = '1';
~S: Outline "O1" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> select * from employees where employee_id = '9999';
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> set noflags;
SQL> set flags 'alternate(lit),nooutline';
SQL> create outline o2 from (select * from employees where employee_id = '1');
SQL>
SQL> set flags 'strat';
SQL> select * from employees where employee_id = '1';
~S: Outline "O1" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> select * from employees where employee_id = '9999';
~S: Outline "O2" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL>
SQL> set flags 'noalt';
SQL> select * from employees where employee_id = '1';
~S: Outline "O1" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> select * from employees where employee_id = '9999';
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> drop outline o1;
SQL> set flags 'alt(literals)';
SQL> select * from employees where employee_id = '1';
~S: Outline "O2" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
SQL> select * from employees where employee_id = '9999';
~S: Outline "O2" used
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMP_EMPLOYEE_ID [1:1]      Direct lookup
0 rows selected
```

As shown in the previous example, Rdb will try to locate an outline using the more generic identifier only if the ALTERNATE_OUTLINE_ID flag has been set.

The ALTERNATE_OUTLINE_ID flag is not set by default and must be explictly set using either SET FLAGS or the RDMS$SET_FLAGS logical.

Any query outline created outside the influence of ALTERNATE_OUTLINE_ID will continue to work because Rdb will use the full signature if no alternate is found.

- The following list of keywords can be specified for the SET FLAGS statement.
  - ALTERNATE_OUTLINE_ID
    The optimizer discards literal values when producing an outline identifier.
  - AUTO_INDEX
    Allows CREATE TABLE and ALTER TABLE to create indices for any PRIMARY KEY, FOREIGN KEY or UNIQUE constraint added to the table.
  - AUTO_OVERRIDE
    Allows a user with the DBADM (administrator) privilege to insert or update a column defined as AUTOMATIC.
  - BITMAPPED_SCAN
    Enables use of in-memory compressed DBKEY bitmaps for index AND and OR operations in the dynamic optimizer.
  - BLR
    Displays the binary language (BLR) representation request for the query.
  - CARDINALITY
    Shows cardinality updates.
  - CARTESIAN_LIMIT
    Limits the number of small tables that are allowed to be placed anywhere in the join order.
  - CHRONO_FLAG
    Display a timestamp before debug dump output. An optional value can be enclosed in parentheses and can be 0, 1, or 2.

    CHRONO_FLAG(0) and NOCHRONO_FLAG are equivalent. If you specify CHRONO_FLAG but omit n, the default is CHRONO_FLAG(1). CHRONO_FLAG(1) enables an additional trace message that includes the attach number and the current time. CHRONO_FLAG(2) enables an additional trace message that includes the attach number and the current date and time. If you supply a value for n that is greater than 2, it is ignored, and a value of 1 is used.
  - CONTROL_BITS
    Displays a decoding of the BLR$K_CONTROL_BITS semantic flags when used with the BLR keyword.
  - COSTING
    Displays traces on optimizer costing.
  - COUNT_SCAN
    Enables count scan optimization on sorted and sorted ranked indexes, where the optimizer will use cardinality information from the index to efficiently determine the count of rows that satisfy the query.
  - CURSOR_STATS
    Displays general cursor statistics for the optimizer.
  - DATABASE_PARAMETERS
    Displays the database parameter buffer during ATTACH, CREATE DATABASE, ALTER DATABASE, IMPORT DATABASE, and DISCONNECT statements.

* DDAL
  This flag allows a knowledgeable database administrator to override the normal prohibition that system indices cannot be created or dropped when creating special indices for Replication Option for Rdb.
* DDL_BLR
  Displays the binary language (BLR) representation of expressions within data definitions, such as the expression for a computed column within a table definition.
* DETAIL_LEVEL
  A debug flag used with other debug flags to enable additional detailed information in the debug output. The DETAIL_LEVEL keyword can be followed by a numeric value in parentheses. For those debug flags that support it, this indicates the degree of additional detail to be shown.
* ESTIMATES
  Displays the optimizer estimates.
* EXECUTION
  Displays an execution trace from the dynamic optimizer. For a sequential retrieval from a table that is strictly partitioned, this includes a count and a list of the selected partitions each time the query executes.

  The EXECUTION keyword can be followed by a numeric value in parentheses. This represents the number of lines to display before stopping the execution trace for the query execution. There can be no spaces between the keyword and the parameter. The default is 100.
* HASHING
  Enables all query HASHING optimizations. Use specific parameters to control various hashing optimizations.

  See also the NOHASHING keyword.

Note

Other keywords are reserved for future use by Oracle.

* HASHING (JOINS)
  The keyword JOINS enables the HASHING for table joins.

* IGNORE_OUTLINE
  Ignores outlines defined in the database. The IGNORE_OUTLINE keyword has the same action as setting the RDMS$BIND_OUTLINE_FLAGS logical name to 1.
* INDEX_COLUMN_GROUP
  Enables leading index columns as workload column groups. This may increase solution cardinality accuracy.
* INDEX_DEFER_ROOT
  When this flag is set and an index is created, the index root node is created in the database only when there is data in the table. If the table is empty, creation of the index root node is deferred until rows are inserted into the table.

* INDEX_PARTITIONS
  Displays index partitioning information as part of a dynamic execution trace.
* INDEX_SIZING (LARGE)
  Enables the large node algorithm for default NODE SIZE computation. This has the same effect as defining the logical name RDMS$DEFAULT_INDEX_NODE_SIZE_SMALL as false (0). This is the default behavior and is equivalent to NOINDEX_SIZING.
* INDEX_SIZING (SMALL)
  Enables the small node algorithm for default NODE SIZE computation. This has the same effect as defining the logical name RDMS$DEFAULT_INDEX_NODE_SIZE_SMALL as true (1).
* INDEX_STATS
  Enables debug flags output for the progress of an ALTER, CREATE, or DROP INDEX statement.
* INTERNALS
  Enables debug flags output for internal queries such as constraints and triggers. It can be used in conjunction with other keywords such as STRATEGY, BLR, and EXECUTION.
* ITEM_LIST
  Displays item list information passed in for the database queries and as compile-time query options.
* JOIN_ORDER
  Displays the order of joining each join item for each query.
* LAREA_READY
  This flag can be used to investigate table and index locking behavior. This flag is disabled by default.
* MAX_RECURSION
  Sets the maximum number of recursions that can be performed when executing a match strategy. This prevents excessive recursion in the processing of the match strategy. The default value is 100. The equivalent logical name is RDMS$BIND_MAX_RECURSION.
* MAX_SOLUTION
  Enables maximum search space for possible retrieval solutions. If enabled, the optimizer will try more solutions based on each leading segment of the index, and thus may create more solutions than before, but may find more efficient solutions applying multiple segments in index retrieval. The equivalent logical name is RDMS$DISABLE_MAX_SOLUTION. The default is enabled.
* MAX_STABILITY
  Enables maximum stability; the dynamic optimizer is not allowed. The MAX_STABILITY keyword has the same action as the RDMS$MAX_STABILITY logical name.
* MBLR
  Displays the metadata binary language
  representation request for the data definition language statement.

* MODE
Allows you to specify which query outline should be used by specifying the mode value of that query outline. An optional value can be enclosed in parentheses and can be any positive or negative integer.

If you specify MODE but omit a value, the default is MODE(1). If you specify MODE(0) or NOMODE, it disables the display of the mode in the SHOW FLAGS statement output. MODE(0) is the default for Oracle Rdb generated outlines.

* NOHASHING
Disables all query HASHING optimizations. Use specific parameters to control various hashing optimizations.

See also the HASHING keyword.

Note

Other keywords are reserved for future use by Oracle.

---

* NOHASHING (JOINS)
The keyword JOINS disables the HASHING for table joins.

* NOINDEX_SIZING
Enables the large node algorithm for default NODE SIZE computation. This has the same effect as defining the logical name RDMS$DEFAULT_INDEX_NODE_SIZE_SMALL as false (0). This is the default behavior and is equivalent to INDEX_SIZING(LARGE).

* NONE
Used to turn off all currently defined keywords. Equivalent to SET NOFLAGS.

* NOREWRITE
Disables all query rewrite optimizations. Use specific parameters to control various query rewrite optimizations.

See also the REWRITE keyword.

* NOREWRITE (BOOLEANS)
Specifying the BOOLEANS keyword will disable only the BOOLEANS predicate rewrite optimization.

* NOREWRITE (CONTAINING)
Specifying the CONTAINING keyword will disable only the CONTAINING predicate rewrite optimization.

* NOREWRITE (IN_CLAUSE)
Specifying the IN_CLAUSE keyword will disable only the IN_CLAUSE predicate rewrite optimization.

* NOREWRITE (IS_NULL)
Specifying the IS_NULL keyword will disable only the IS_NULL predicate rewrite optimization.

* NOREWRITE (LIKE)
Specifying the LIKE keyword will disable only the LIKE predicate rewrite optimization.

* NOREWRITE (ORDERING)
  Specifying the ORDERING keyword will disable the ordering rewrite optimization.
* NOREWRITE (STARTING_WITH)
  Specifying the STARTING_WITH keyword will disable only the STARTING WITH predicate rewrite optimization.
* NOREWRITE (UNKNOWN)
  Specifying the UNKNOWN keyword will disable only the UNKNOWN predicate rewrite optimization.
* OBLR
  Displays query outline in Binary Language Representation (BLR).
* OLD_COST_MODEL
  Enables the old cost model. The OLD_COST_MODEL keyword has the same action as the RDMS$USE_OLD_COST_MODEL logical name.
* OPTIMIZATION_LEVEL
  Used to change the default optimization level for a query. If the query explicitly uses the OPTIMIZE FOR clause, or is compiled within an environment which overrides the default using a method such as SET OPTIMIZATION LEVEL, then no change will occur. If the query uses the default optimization level, then the optimization will be modified by this flag. With no option specified or an empty options list, this will default to TOTAL TIME. The flag NOOPTIMIZATION_LEVEL will revert to the default Oracle Rdb behavior.
* OPTIMIZATION_LEVEL (FAST_FIRST)
  Sets FAST FIRST as the default optimization level for queries in all sessions.
* OPTIMIZATION_LEVEL (TOTAL_TIME)
  Sets TOTAL TIME as the default optimization level for queries in all sessions.
* OUTLINE
  Displays query outline for this query (can be used without STRATEGY keyword).
* PREFIX
  Used with BLR keyword to inhibit offset numbering and other formatting of binary language representation display.
* QUERY_CPU_TIME_LIMIT
  Defines the limit on the CPU time consumed by the query compiler. Equivalent to logical name RDMS$BIND_QG_CPU_TIMEOUT.
* QUERY_ROWS_LIMIT
  Defines the limit on the number of rows returned to an application. Equivalent to logical name RDMS$BIND_QG_REC_LIMIT.
* QUERY_TIME_LIMIT
  Defines the limit on the time consumed by the query compiler. Equivalent to logical name RDMS$BIND_QG_TIMEOUT.

* REBUILD_SPAM_PAGES
  The flag REBUILD_SPAM_PAGES is for use in conjunction with the DDL commands ALTER TABLE, ALTER STORAGE MAP, and ALTER INDEX. When changing the row length or THRESHOLDS clause for a table or index, the corresponding SPAM pages for the logical area may require rebuilding. By default, these DDL commands update the AIP and set a flag to indicate that the SPAM pages should be rebuilt. However, this flag may be set prior to executing a COMMIT for the transaction and the rebuild will take place within this transaction. Use SET FLAGS 'NOREBUILD_SPAM_PAGES' to negate this flag.
* REFINE_ESTIMATES
  Controls index estimation in the dynamic optimizer.
* REQUEST_NAMES
  Displays the names of user requests, triggers, and constraints.
* REVERSE_SCAN
  Enables the reverse index scan strategy. The NOREVERSE_SCAN keyword has the same action as the RDMS$DISABLE_REVERSE_SCAN logical name.
* REWRITE
  Enables all query rewrite optimizations. Use specific parameters to control various query rewrite optimizations.

  See also the NOREWRITE keyword.
* REWRITE (BOOLEANS)
  Specifying the BOOLEANS keyword will enable only the BOOLEANS predicate rewrite optimization. Oracle Rdb will propagate the TRUE, FALSE and UNKNOWN boolean values through the query predicate; removing FALSE branches from the query.
* REWRITE (CONTAINING)
  Specifying the CONTAINING keyword will enable only the CONTAINING predicate rewrite optimization.
* REWRITE (IN_CLAUSE)
  Specifying the IN_CLAUSE keyword will enable only the IN_CLAUSE predicate rewrite optimization. When there are no NULL values in the tables, Oracle Rdb can rewrite the query replacing NOT IN with the more efficient EXISTS clause.
* REWRITE (IS_NULL)
  Specifying the IS_NULL keyword will enable only the IS_NULL predicate rewrite optimization. Oracle Rdb will apply knowledge of the NOT NULL and PRIMARY KEY state of the column to the IS NULL and IS NOT NULL predicates.
* REWRITE (LIKE)
  Specifying the LIKE keyword will enable only the LIKE predicate rewrite optimization.
* REWRITE (ORDERING)
  Specifying the ORDERING keyword will enable only the ordering rewrite optimization. Oracle Rdb will apply knowledge of the constant nature of expressions to remove them from the DISTINCT, ORDER BY, GROUP BY, UNION and similar cases where the constant or invariant values have no

effect on the ordering of the results. The benefit is smaller memory used when ordering data and possible elimination of the ORDER BY action.

* REWRITE (STARTING_WITH)
  Specifying the STARTING_WITH keyword will enable only the STARTING WITH predicate rewrite optimization.
* REWRITE (UNKNOWN)
  Specifying the UNKNOWN keyword will enable only the UNKNOWN predicate rewrite optimization. Oracle Rdb will replace predicates that compare values with NULL with the UNKNOWN boolean value.
* SCROLL_EMULATION
  Disables scrolling for old-style LIST OF BYTE VARYING (segmented string) format. The SCROLL_EMULATION flag has the same action as setting the RDMS$DIAG_FLAGS logical name to L.
* SELECTIVITY
  Refers to the methods by which the static optimizer estimates predicate selectivity. This flag takes a numeric value in parentheses from 0 to 3.

  0 = standard (non-aggressive, non-sampled) selectivity
  1 = aggressive and non-sampled selectivity
  2 = sampled and non-aggressive selectivity
  3 = sampled and aggressive selectivity.

  By default the flag is disabled, which is the equivalent of setting its value to 0.
* SEQ_CACHE
  Adjusts the sequence cache size for the process issuing the SET FLAGS statement. An optional value can be enclosed in parentheses and must be a numeric value greater than 2.

  Specifying a value of 1 is equivalent to specifying NOSEQ_CACHE.
  Use SEQ_CACHE to override the CACHE setting for all sequences subsequently referenced by the application. The new cache size does not affect any sequence that has already been referenced, or any sequence defined as NOCACHE.
* SOLUTIONS
  Displays traces on optimizer solutions.
* SORTKEY_EXT
  Reports if ORDER BY (or SORTED BY) is referencing only external (constant) value. The SORTKEY_EXT flag has the same action as setting the RDMS$DIAG_FLAGS logical name to S.
* SORT_STATISTICS
  Displays sort statistics during execution.
* STOMAP_STATS
  Displays the processing of storage maps for any tables that are altered, dropped or affected by the ALTER DATABASE ... DROP STORAGE AREA ... CASCADE statement.
* STRATEGY
  Shows the optimizer strategy. If a table is strictly partitioned, the text "(partitioned scan#nn)" appears after the table name, where nn indicates the leaf number for a sequential scan (there may be several within a single query).

* TEST_SYSTEM
  This flag is used by the Oracle Rdb testing environment to modify the output of various functions, trace and debugging displays. It is used to eliminate data in test output that would normally cause differences between test executions.
* TRACE
  Enables output from TRACE statement. See also PREFIX which affects the output of the TRACE Statement.
* TRANSACTION_PARAMETERS
  Displays the transaction actions (and parameter buffer) during the execution of SAVEPOINT, SET TRANSACTION, START TRANSACTION, COMMIT, COMMIT AND CHAIN, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK AND CHAIN, and ROLLBACK TO SAVEPOINT Statements.
* TRANSITIVITY
  Enables transitivity between selections and join predicates. The NOTRANSITIVITY keyword has the same action as the RDMS$DISABLE_TRANSITIVITY logical name.
* VALIDATE_ROUTINE
  Enables revalidation of an invalidated stored procedure or stored function. The VALIDATE_ROUTINE keyword has the same action as the RDMS$VALIDATE_ROUTINE logical name.
* VARIANCE_DOF
  Sets the default degree of freedom (DOF) for calculation of the mean (average) in small samples (instead of using the VARIANCE function). Only the values 0 and 1 are allowed. This keyword is deprecated; instead use the SQL functions VAR_POP, VAR_SAMP, STDDEV_POP or STDDEV_SAMP for these semantics.
* WARN_DDL
  Sometimes legal data definitions can have side effects, this flag allows these warning to be enabled and disabled. This flag is enabled by default, with the exception for when attached by PATHNAME. The data definition statement still succeeds even with the reported warnings. The database administrator may choose to rollback the statement based on this feedback.
* WARN_INVALID
  Reports invalidated objects during the ALTER INDEX, DROP INDEX, DROP TABLE, and DROP MODULE statements.
* WATCH_CALL
  Traces the execution of queries, triggers and stored functions and procedures. The output includes the name of the trigger, function or procedure or "unnamed" for an anonymous query. In most cases a query can be named using the OPTIMIZE AS clause. It also includes the value of CURRENT_USER during the execution of that routine. CURRENT_USER may be inherited from any module that uses the AUTHORIZATION clause.
* WATCH_OPEN
  Traces all queries executed on the database. This may include SQL runtime system queries to lookup table names, etc as well as queries executed by the application. The output includes the 32 digit hex

identifier, the same as that used by the CREATE OUTLINE statement. This value uniquely identifies the query being executed.

If a query is a stored routine (function or procedure) then the notation "(stored)" is appended, if the query is named then it will be classified as "(query)", otherwise it will be designated as "(unnamed)".

* ZIGZAG_MATCH
  Enables zigzag key skip on both outer and inner match loops. When you specify the ZIGZAG_MATCH keyword with the NOZIGZAG_OUTER keyword, it disables zigzag key skip on the outer loop (and has the same action as setting the RDMS$DISABLE_ZIGZAG_MATCH logical name to 1). The NOZIGZAG_MATCH keyword disables zigzag key skip on both outer and inner match loops (and has the same action as setting the RDMS$DISABLE_ZIGZAG_MATCH logical name to 2).
* ZIGZAG_OUTER
  Enables zigzag key skip on the outer loop. See the entry for ZIGZAG_MATCH for information on the action taken when you specify ZIGZAG_OUTER and ZIGZAG_MATCH together.

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
```

Example 3: Using Host Variables in Interactive SQL

The example also demonstrates using literal strings with multiple options to enable and disable flags.

```
SQL> SHOW FLAGS
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX
SQL> -- declare a host variable to be used with SET FLAGS
SQL> declare :hv char(40);
SQL> -- assign a value to the variable
SQL> begin
cont> set :hv = 'strategy, outline';
cont> end;
SQL> -- use the host variable to enable or disable flags
SQL> set flags :hv;
SQL> show flags
```

```
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   STRATEGY,PREFIX,OUTLINE
SQL> -- use a string literal directly with the SET FLAGS statement
SQL> set flags 'noprefix,execution(10)';
SQL> show flags

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   STRATEGY,OUTLINE,EXECUTION(10)
```

Example 4: Using the MODE(n) Flag

```
SQL> SET FLAGS 'MODE(10), OUTLINE';
SQL> SHOW FLAGS
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,OUTLINE,MODE(10)
SQL> SELECT COUNT(*) FROM EMPLOYEES;
-- Rdb Generated Outline : 30-MAY-1997 16:35
create outline QO_B3F54F772CC05435_0000000A
id 'B3F54F772CC054350B2B454D95537995'
mode 10
as (
  query (
-- For loop
    subquery (
      subquery (
        EMPLOYEES 0     access path index       EMP_EMPLOYEE_ID
        )
      )
    )
  )
compliance optional     ;
         100
1 row selected
```

Example 5: Using the WARN_INVALID Debug Flag

```
SQL> SET FLAGS 'WARN_INVALID';
SQL> SHOW FLAGS;
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,WARN_INVALID
SQL> -- warning because of dependencies
SQL> DROP TABLE T1 CASCADE;
~Xw: Routine "P3" marked invalid
~Xw: Routine "P2" marked invalid
~Xw: Routine "P1" marked invalid
SQL>
SQL> -- Create an outline that references an INDEX.
SQL> CREATE TABLE T1 (A INTEGER, B INTEGER);
SQL> CREATE INDEX I1 ON T1 (A);
SQL> CREATE OUTLINE QO1
cont> ID '19412AB61A7FE1FA6053F43F8F01EE6D'
cont> MODE 0
cont> AS (
cont>   QUERY (
cont>     SUBQUERY (
cont>       T1 0    ACCESS PATH INDEX       I1
cont>       )
cont>     )
cont>   )
cont> COMPLIANCE OPTIONAL;
SQL>
SQL> -- Warning because of disabled index
SQL> ALTER INDEX I1
```

```
cont>     MAINTENANCE IS DISABLED;
~Xw: Outline "QO1" marked invalid (index "I1" disabled)
SQL> SHOW OUTLINE QO1;
     QO1
    Object has been marked INVALID
 Source:
CREATE OUTLINE QO1
ID '19412AB61A7FE1FA6053F43F8F01EE6D'
MODE 0
AS (
  QUERY (
    SUBQUERY (
      T1 0    ACCESS PATH INDEX       I1
      )
    )
  )
COMPLIANCE OPTIONAL;
```

Example 6: Using the INTERNAL Keyword to Display Trigger Actions

```
SQL> -- The following code shows the strategy used by the trigger
SQL> -- actions on the AFTER DELETE trigger on EMPLOYEES
SQL> SET FLAGS 'STRATEGY, INTERNALS, REQUEST_NAMES';
SQL> SHOW FLAGS
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   INTERNALS,STRATEGY,PREFIX,REQUEST_NAMES
SQL> DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = '00164';
~S: Trigger name  EMPLOYEE_ID_CASCADE_DELETE
Get     Temporary relation      Retrieval by index of relation DEGREES
  Index name  DEG_EMP_ID [1:1]
~S: Trigger name  EMPLOYEE_ID_CASCADE_DELETE
Get     Temporary relation      Retrieval by index of relation JOB_HISTORY
  Index name  JOB_HISTORY_HASH [1:1]
~S: Trigger name  EMPLOYEE_ID_CASCADE_DELETE
Get     Temporary relation      Retrieval by index of relation SALARY_HISTORY
  Index name  SH_EMPLOYEE_ID [1:1]
~S: Trigger name  EMPLOYEE_ID_CASCADE_DELETE
Conjunct        Get     Retrieval by index of relation DEPARTMENTS
  Index name  DEPARTMENTS_INDEX [0:0]
Temporary relation      Get     Retrieval by index of relation EMPLOYEES
  Index name  EMPLOYEES_HASH [1:1]       Direct lookup
1 row deleted
```

Example 7: Using the INDEX_COLUMN_GROUP Keyword

```
SQL> -- The table STUDENTS has an index on the two columns
SQL> -- STU_NUM and COURSE_NUM. When the INDEX_COLUMN_GROUP
SQL> -- keyword is not set, the optimizer uses a fixed
SQL> -- proportion of the table cardinality based on the equality
SQL> -- with the STU_NUM column. In this example, 5134 rows are expected,
SQL> -- when in reality, only 9 are returned by the query.
SQL> CREATE INDEX STUDENT_NDX ON STUDENTS (STU_NUM,COURSE_NUM DESC);
SQL> --
SQL> SELECT STU_NUM FROM STUDENTS
cont>  WHERE STU_NUM = 191270771
cont>  ORDER BY OTHER_COLUMN;
Solutions tried 2
Solutions blocks created 1
Created solutions pruned 0
Cost of the chosen solution   4.5644922E+03
Cardinality of chosen solution   5.1342500E+03
```

```
~O: Physical statistics used
Sort
SortId# 7., # Keys 2
  Item# 1, Dtype: 2, Order: 0, Off: 0, Len: 1
  Item# 2, Dtype: 35, Order: 0, Off: 1, Len: 8
  LRL: 32, NoDups:0, Blks:327, EqlKey:0, WkFls: 2
Leaf#01 BgrOnly STUDENTS Card=164296
  BgrNdx1 STUDENT_NDX [1:1] Fan=14
   191270771
   191270771
   191270771
   191270771
   191270771
   191270771
   191270771
   191270771
SORT(9) SortId# 7, --------------------- Version: V5-000
  Records Input: 9      Sorted: 9       Output: 0
  LogRecLen Input: 32     Intern: 32      Output: 32
  Nodes in SoTree: 5234   Init Dispersion Runs: 0
  Max Merge Order: 0      Numb.of Merge passes: 0
  Work File Alloc: 0
  MBC for Input: 0        MBC for Output: 0
  MBF for Input: 0        MBF for Output: 0
  Big Allocated Chunk: 4606464 busy
   191270771
9 rows selected
SQL> --
SQL> -- When you use the SET FLAGS statement to set the
SQL> -- INDEX_COLUMN_GROUP keyword, it activates the optimizer
SQL> -- to consider the index segment columns as a workload column
SQL> -- group, compute the statistics for duplicity factor and null
SQL> -- factor dynamically, and then apply them in estimating the
SQL> -- cardinality of the solution.
SQL> --
SQL> SET FLAGS 'INDEX_COLUMN_GROUP';
SQL> -- The following is the optimizer cost estimate and sort output trace
SQL> -- for the previous query with INDEX_COLUMN_GROUP enabled. The optimizer
SQL> -- now estimates a lower cardinality of about 8 rows.
Solutions tried 2
Solutions blocks created 1
Created solutions pruned 0
Cost of the chosen solution   3.8118614E+01
Cardinality of chosen solution   8.3961573E+00
~O: Workload and Physical statistics used
Sort
SortId# 2., # Keys 2
  Item# 1, Dtype: 2, Order: 0, Off: 0, Len: 1
  Item# 2, Dtype: 35, Order: 0, Off: 1, Len: 8
  LRL: 32, NoDups:0, Blks:7, EqlKey:0, WkFls: 2
Leaf#01 BgrOnly STUDENTS Card=164296
  BgrNdx1 STUDENT_NDX [1:1] Fan=14
   191270771
   191270771
   191270771
   191270771
   191270771
   191270771
   191270771
   191270771
SORT(2) SortId# 2, --------------------- Version: V5-000
  Records Input: 9      Sorted: 9       Output: 0
LogRecLen Input: 32     Intern: 32      Output: 32
Nodes in SoTree: 114    Init Dispersion Runs: 0
Max Merge Order: 0      Numb.of Merge passes: 0
```

```
Work File Alloc: 0
MBC for Input: 0        MBC for Output: 0
MBF for Input: 0        MBF for Output: 0
Big Allocated Chunk: 87552 idle
   191270771
9 rows selected
```

Example 8: Using the AUTO_OVERRIDE Keyword

```
SQL> -- Suppose that after year 2000 testing was performed on a
SQL> -- production system, the system date and time were not reset
SQL> -- to the correct date.  This was not noticed until
SQL> -- after transactions for a full day had been stored. To
SQL> -- correct this problem, the database administrator overrides
SQL> -- the READ ONLY characteristic of the AUTOMATIC column and
SQL> -- adjusts the date and time.
SQL> SELECT * FROM ACCOUNTS
cont> WHERE LAST_UPDATE > DATE'2001-1-1';
       ACCOUNT_NO     LAST_NAME          LAST_UPDATE    CURRENT_BALANCE
       NULL           Smith              2001-06-02      100000.000
1 row selected
SQL> -- Attempts to fix the date and time fail because the
SQL> -- column is AUTOMATIC.
SQL> UPDATE ACCOUNTS
cont>     SET LAST_UPDATE = LAST_UPDATE - INTERVAL'1' YEAR
cont>     WHERE LAST_UPDATE > DATE'2000-1-1';
%RDB-E-READ_ONLY_FIELD, attempt to update the read-only field LAST_UPDATE
SQL> --
SQL> SET FLAGS 'AUTO_OVERRIDE';
SQL> SHOW FLAGS
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,AUTO_OVERRIDE
SQL>--
SQL> -- Fix the date and time.
SQL> UPDATE ACCOUNTS
cont>     SET LAST_UPDATE = LAST_UPDATE - INTERVAL'1' YEAR
cont>     WHERE LAST_UPDATE > DATE'2000-1-1';
1 row updated
SQL>
SQL> SELECT * FROM ACCOUNTS;
           ACCOUNT_NO   LAST_NAME              LAST_UPDATE   CURRENT_BALANCE
                 NULL   Smith                  1999-06-02         100000.000
1 row selected
SQL>
SQL> SET FLAGS 'NOAUTO_OVERRIDE';
```

Example 9: Using the AUTO_INDEX option

```
SQL> set dialect 'SQL92';
SQL> set flags 'AUTO_INDEX,INDEX_STATS';
SQL> create table PERSON
cont> (employee_id      integer primary key,
cont>  manager_id       integer references PERSON (employee_id),
cont>  last_name        char(30),
cont>  first_name       char(30),
cont>  unique (last_name, first_name));
~Ai create index "PERSON_PRIMARY_EMPLOYEE_ID"
~Ai larea length is 430
~Ai storage area (default) larea=57
~Ai create sorted index, ikey_len=5
Sort    Get     Retrieval sequentially of relation PERSON
~Ai create index partition, node=430 %fill=0
~Ai create index "PERSON_FOREIGN1"
~Ai larea length is 215
~Ai storage area is shared: larea=57
~Ai create sorted index, ikey_len=5
Sort    Get     Retrieval sequentially of relation PERSON
~Ai create index partition, node=0 %fill=0
~Ai create index "PERSON_UNIQUE1"
~Ai larea length is 215
~Ai storage area is shared: larea=57
~Ai create sorted index, ikey_len=62
Sort    Get     Retrieval sequentially of relation PERSON
~Ai create index partition, node=0 %fill=0
SQL>
SQL> show table (index) person
Information for table PERSON

Indexes on table PERSON:
PERSON_FOREIGN1                 with column MANAGER_ID
  Duplicates are allowed
  Type is Sorted
  Key suffix compression is DISABLED

PERSON_PRIMARY_EMPLOYEE_ID      with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Sorted
  Key suffix compression is DISABLED
  Node size  430

PERSON_UNIQUE1                  with column LAST_NAME
                                and column FIRST_NAME
  Duplicates are allowed
  Type is Sorted
  Key suffix compression is DISABLED
SQL>
```

Example 10: Using the WATCH_CALL option

This example shows the output of WATCH_CALL for an INSERT statement which causes an AFTER INSERT trigger (AFTER_INSERT) to be executed which calls an SQL function WRITE_TEXT to trace the input data. It then traces a query named using OPTIMIZE AS clause.

```
SQL> insert into SAMPLE_T values ('Fred');
~Xa: routine "(unnamed)", user=SMITH
~Xa: routine "AFTER_INSERT", user=SMITH
~Xa: routine "WRITE_TEXT", user=SMITH
~Xt: Fred
1 row inserted
SQL> select * from SAMPLE_T
cont>     optimize as LOOKUP_SAMPLE_T;
~Xa: routine "LOOKUP_SAMPLE_T", user=SMITH
 NEW_NAME
 Fred
1 row selected
```

Example 11: Using the WATCH_OPEN option

This example shows the output of WATCH_OPEN for the same INSERT statement as seen in example 10.

```
SQL> insert into SAMPLE_T values ('Fred');
~Xo: Start Request B667E51E3625026EB7FFF3F4D3A16DC3 (unnamed)
~Xo: Start Request A8568053FE5A1A0852A1BE83A884016F "AFTER_INSERT" (query)
~Xo: Start Request 08AE59062657299B4768F6C2DFB6928E "WRITE_TEXT" (stored)
~Xt: Fred
1 row inserted
SQL>
SQL> select * from SAMPLE_T
cont>     optimize as LOOKUP_SAMPLE_T;
~Xo: Start Request F6025FAB1DD36B0DE0E52F3A9641BC5F "LOOKUP_SAMPLE_T" (query)
 NEW_NAME
 Fred
 Fred
2 rows selected
```

Example 12: Using SET FLAGS from an application program

The SET FLAGS statement can be executed from Dynamic SQL using one of two methods.

- The first method is immediate execution by passing a string literal. The string literal argument to SET FLAGS requires that the single quote marks be doubled for correct inclusion in the string literal argument to EXECUTE IMMEDIATE.
- The second method is to pass the entire SET FLAGS statement in a parameter to EXECUTE IMMEDIATE

```
exec sql
    execute immediate 'set flags ''strategy''';
```

The entire SET FLAGS statement could be in a parameter to EXECUTE IMMEDIATE

```
exec sql
    execute immediate :set_flags_text;
```

If SET FLAGS is executed multiple times it can be prepared as a dynamic statement (PREPARE) and then the statement name used for multiple executions. The input marker (?) is substituted on different calls to EXECUTE the previously prepared statement.

```
#include <string.h>
#include <sql_rdb_headers.h>

void main ()
{
int SQLCODE;
char myflags[40];

exec sql
    prepare set_flags_stmt from 'set flags ?';
if (SQLCODE != 0)
    sql_signal ();

strcpy (myflags, "transaction,item_list");
exec sql
    execute set_flags_stmt using :myflags;
if (SQLCODE != 0)
    sql_signal ();

exec sql
    start transaction;
if (SQLCODE != 0)
    sql_signal ();

strcpy (myflags, "notransaction,noitem_list");
exec sql
    execute set_flags_stmt using :myflags;
if (SQLCODE != 0)
    sql_signal ();

exec sql
    rollback;
if (SQLCODE != 0)
    sql_signal ();
}
```

Example 13: Using the CHRONO_FLAG option

The use of CHRONO_FLAG adds more timestamp information to the log.

```
SQL> set flags 'chrono_fla(2),transaction';
SQL> start transaction;
 ATTACH #1, 29-NOV-2003 10:08:37.51
~T Compile transaction (1) on db: 1
~T Transaction Parameter Block: (len=2)
0000 (00000) TPB$K_VERSION = 1
0001 (00001) TPB$K_WRITE (read write)
 ATTACH #1, 29-NOV-2003 10:08:37.58
~T Start_transaction (1) on db: 1, db count=1
SQL> rollback;
 ATTACH #1, 29-NOV-2003 10:08:46.74
~T Rollback_transaction (1) on db: 1
SQL> rollback;
 ATTACH #1, 29-NOV-2003 10:08:46.74
~T Rollback_transaction (1) on db: 1
SQL>
```

Example 14: Using the REBUILD_SPAM_PAGES option

When changing the row length or THRESHOLDS clause for a table or index, the corresponding SPAM pages for the logical area may require rebuilding. By default, these DDL commands update the AIP and set a flag to indicate that the SPAM pages should be rebuilt. However, this flag may be set prior to executing a COMMIT for the transaction and the rebuild will take place within this transaction.

The following example shows a simple change to the EMPLOYEES table (mapped in this example to set of UNIFORM areas). The flag STOMAP_STATS is used to enable more trace information from the ALTER and COMMIT statements.

```
SQL> set transaction read write;
SQL>
SQL> set flags 'stomap_stats';
SQL>
SQL> alter table EMPLOYEES
cont>     add column MANAGERS_COMMENTS varchar(300);
~As: reads: async 0 synch 94, writes: async 18 synch 1
SQL>
SQL> alter storage map EMPLOYEES_MAP
cont>     store
cont>         using (EMPLOYEE_ID)
cont>             in EMPIDS_LOW
cont>           (thresholds (34,76,90))
cont>                 with limit of ('00200')
cont>             in EMPIDS_MID
cont>           (thresholds (34,76,90))
cont>                 with limit of ('00400')
cont>             otherwise in EMPIDS_OVER
cont>           (thresholds (34,76,90));
~As locking table "EMPLOYEES" (PR -> PU)
~As: removing superseded routine EMPLOYEES_MAP
~As: creating storage mapping routine EMPLOYEES_MAP (columns=1)
~As: reads: async 0 synch 117, writes: async 56 synch 0
SQL>
SQL> set flags 'rebuild_spam_pages';
SQL>
SQL> commit;
%RDMS-I-LOGMODVAL,     modified record length to 423
%RDMS-I-LOGMODVAL,     modified space management thresholds to (34%, 76%, 90%)
%RDMS-I-LOGMODVAL,     modified record length to 423
%RDMS-I-LOGMODVAL,     modified space management thresholds to (34%, 76%, 90%)
%RDMS-I-LOGMODVAL,     modified record length to 423
%RDMS-I-LOGMODVAL,     modified space management thresholds to (34%, 76%, 90%)
SQL>
```

The message LOGMODVAL will appear for each logical area in the storage map, one per partition.

This rebuild action only applies to UNIFORM storage areas and may incur significant I/O as SPAM pages and data pages are read to allow the SPAM page to be rebuilt.

Example 15: Using the OPTIMIZATION_LEVEL flag

The following example shows how the behavior of a query changes using the dynamic optimizer with the OPTIMIZATION_LEVEL flag set.

```
SQL> -- show with default behavior (FFirst tactic used)
SQL> select *
cont> from xtest
cont> where col2 between 999980 and 1000000
cont>   and col1 > 0
cont> ;
Tables:
  0 = XTEST
Leaf#01 FFirst 0:XTEST Card=10
  Bool: (0.COL2 >= 999980) AND (0.COL2 <= 1000000) AND (0.COL1 > 0)
  BgrNdx1 XTEST_IDX [1:0] Fan=17
    Keys: 0.COL1 > 0
0 rows selected
SQL>
SQL> -- use SET FLAGS
SQL> set flags 'optimization_level(total_time)';
SQL>
SQL> -- show that BgrOnly is used for TOTAL TIME
SQL> select *
cont> from xtest
cont> where col2 between 999980 and 1000000
cont>   and col1 > 0
cont> ;
Tables:
  0 = XTEST
Leaf#01 BgrOnly 0:XTEST Card=10
  Bool: (0.COL2 >= 999980) AND (0.COL2 <= 1000000) AND (0.COL1 > 0)
  BgrNdx1 XTEST_IDX [1:0] Fan=17
    Keys: 0.COL1 > 0
0 rows selected
SQL>
```

Example 16: Using the ON ALIAS Clause

The default behavior for SET FLAGS is to establish the flag settings on all currently attached databases. This clause will allow the database administrator to set flags on just one database alias.

The following example shows a case where the enabling of AUTO_OVERRIDE required DBADM privilege on the target database but not on the source database. It may be that the current user does not have (or really need) DBADM privilege on that database.

```
SQL> -- Now enable AUTO_OVERRIDE on only one database
SQL> set flags (on alias abc_a) 'auto_override';
SQL> set flags (on alias abc_b) 'none';
SQL> insert into abc_a.SAMPLE_TABLE select * from abc_b.SAMPLE_SOURCE;
SQL> commit;
```

Example 17: Using the NOREWRITE keyword

```
SQL> set line length 70
SQL> show flags;

Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,WARN_DDL,INDEX_COLUMN_GROUP,MAX_SOLUTION,MAX_RECURSION(100)
   ,REWRITE(CONTAINING),REWRITE(LIKE),REWRITE(STARTING_WITH)
   ,REFINE_ESTIMATES(127),NOBITMAPPED_SCAN
SQL>
SQL> set flags 'norewrite';
SQL> show flags;
```

```
Alias RDB$DBHANDLE:
Flags currently set for Oracle Rdb:
   PREFIX,WARN_DDL,INDEX_COLUMN_GROUP,MAX_SOLUTION,MAX_RECURSION(100)
   ,REFINE_ESTIMATES(127),NOBITMAPPED_SCAN
SQL>
```

SET HOLD CURSORS Statement

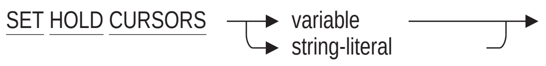
Specifies the session default attributes for holdable cursors that have not been previously defined.

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.
- 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

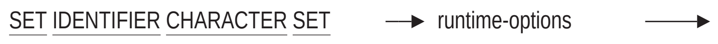
Specifies the identifier character set for the module or interactive SQL session.

Environment

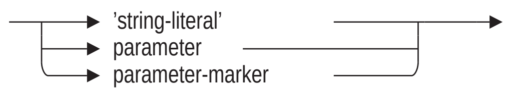
You can use the SET IDENTIFIER CHARACTER SET statement:

- 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

'string-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 Section 2.1.5 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 Section 2.1.5 for a list of allowable character sets.
- If you set the dialect to SQL99 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_KANJI
National character set is DEC_KANJI
Identifier character set is SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
SQL> set identifier character set 'DEC_KANJI';
SQL> show character sets;
Default character set is DEC_KANJI
National character set is DEC_KANJI
Identifier character set is DEC_KANJI
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
```

SET KEYWORD RULES Statement

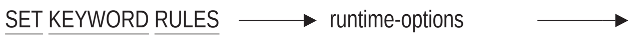
Specifies whether or not you can use identifiers as keywords in the current attach.

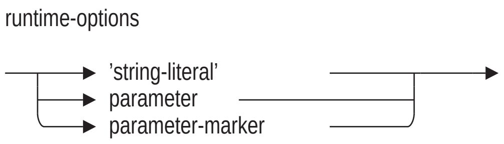
Environment

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





Arguments

'string-literal'
parameter
parameter-marker
Specifies the value of runtime-options, which must be one of the following:

- SQL89, SQL92, SQL99, SQL2011
- MIA
- ORACLE LEVEL1, ORACLE LEVEL2, ORACLE LEVEL3
- SQLV40

Other than SQLV40, all other options force SQL to reject any keyword used as an identifier. The examples show the difference in behavior. The session 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 SQL99

```
SQL> SET KEYWORD RULES 'SQL99';
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

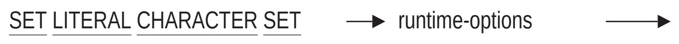
Specifies the literal character set for the module or interactive SQL session.

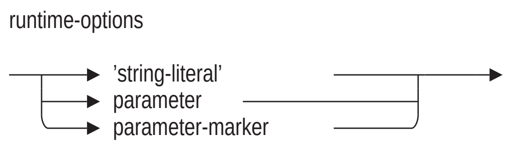
Environment

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

'string-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_KANJI
National character set is DEC_KANJI
Identifier character set is DEC_KANJI
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
SQL> set literal character set 'DEC_KANJI';
SQL> show character sets;
Default character set is DEC_KANJI
National character set is DEC_KANJI
Identifier character set is DEC_KANJI
Literal character set is DEC_KANJI
Display character set is SHIFT_JIS
```

SET NAMES Statement

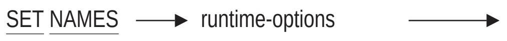
Specifies the default, identifier, and literal character sets for the session. The SET NAMES statement also specifies the character parameters for SQL module language.

Environment

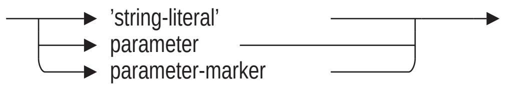
You can use the SET NAMES statement:

- 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

'string-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 Section 2.1.5 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 Section 2.1.5 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
Identifier character set is DEC_MCS
Literal character set is DEC_MCS
Display character set is UNSPECIFIED
SQL> --
SQL> set names 'DEC_KANJI';
SQL> show character sets;
Default character set is DEC_KANJI
National character set is DEC_MCS
Identifier character set is DEC_KANJI
Literal character set is DEC_KANJI
Display character set is UNSPECIFIED
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
Identifier character set is DEC_KANJI
Literal character set is DEC_KANJI
Display character set is UNSPECIFIED
SQL>
```

SET NATIONAL CHARACTER SET Statement

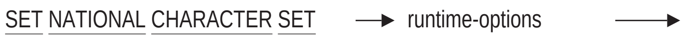
Specifies the national character set for the module or interactive SQL session.

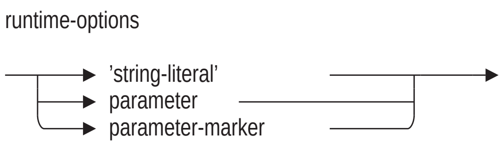
Environment

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





Arguments

'string-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 SQL99 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_KANJI
National character set is DEC_MCS
Identifier character set is SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
SQL> set national character set 'DEC_KANJI';
SQL> show character sets;
Default character set is DEC_KANJI
National character set is DEC_KANJI
Identifier character set is SHIFT_JIS
Literal character set is SHIFT_JIS
Display character set is SHIFT_JIS
```

SET OPTIMIZATION LEVEL Statement

Allows the current session defaults to be specified for query optimization characteristics.

This statement can reset the session defaults using DEFAULT, or can specify one or more keywords for SELECTIVITY or FAST FIRST or TOTAL TIME optimization.

This statement affects all subsequent query compiles in interactive SQL, or queries specified using dynamic SQL.

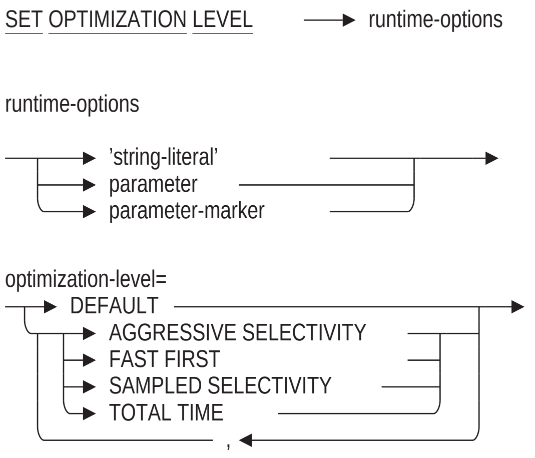
See Chapter 3 and Chapter 4 for information on setting the optimization level in SQL module and precompiler languages.

Environment

You can use the SET OPTIMIZATION LEVEL statement:

- In interactive SQL
- Embedded in host language programs to be precompiled
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

optimization-level
Specifies the optimizer strategy to be used to reset session defaults. The passed string or parameter value must be a formatted list of keyword values. Select from the following options:

- AGGRESSIVE SELECTIVITY option if you expect a small number of rows to be selected.
- DEFAULT option to accept the Oracle Rdb defaults: FAST FIRST and DEFAULT SELECTIVITY.
- FAST FIRST option if you want your program to return data to the user as quickly as possible, even at the expense of total throughput.

- SAMPLED SELECTIVITY option to use literals in the query to perform preliminary estimation on indexes.
- TOTAL TIME option if you want your program to run at the fastest possible rate, returning all the data as quickly as possible. If your application runs in batch, accesses all the records in a query, and performs updates or writes reports, you should specify TOTAL TIME.

Only one of the TOTAL TIME or FAST FIRST options can be selected. Only one of the AGGRESSIVE SELECTIVITY or SAMPLED SELECTIVITY options can be selected. Use a comma to separate the keywords and enclose the list in parentheses. No other options may be included if DEFAULT is selected.

′string-literal′
parameter
parameter-marker
Specifies the value of the runtime-options, which must be a list of keywords, separated by commas.

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 an OPTIMIZE clause in the UPDATE, INSERT, DELETE, or SELECT statement.
- Any query that explicitly includes an OPTIMIZE WITH or OPTIMIZE FOR clause is not affected by the settings established using the SET OPTIMIZATION LEVEL command.

Example

Example 1: Setting the optimization level

The dynamic optimizer can use either FAST FIRST or TOTAL TIME tactics to return rows to the application. The default setting, FAST FIRST, assumes that applications, especially those using interactive SQL, will want to see rows as quickly as possible and possibly abort the query before completion. Therefore, if the FAST FIRST tactic is possible the optimizer will sacrifice overall retrieval time to initially return rows quickly. This choice can be affected by setting the OPTIMIZATION LEVEL.

The following example contrasts the query strategies selected when FAST FIRST versus TOTAL TIME is in effect. Databases and queries will vary in their requirements. Queries should be tuned to see which setting best suits the needs of the application environment. For the MF_PERSONNEL database there is little or no difference between these tactics, but for larger tables the differences could be noticeable.

```
SQL> set flags 'STRATEGY,DETAIL';
SQL> --
SQL> -- No optimization level has been selected. The optimizer
SQL> -- selects the FAST FIRST (FFirst) retrieval tactic 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');
Tables:
  0 = EMPLOYEES
Leaf#01 FFirst 0:EMPLOYEES Card=100
  Bool: (0.EMPLOYEE_ID = '00167') OR (0.EMPLOYEE_ID = '00168')
  BgrNdx1 EMPLOYEES_HASH [(1:1)2] Fan=1
    Keys: r0: 0.EMPLOYEE_ID = '00168'
          r1: 0.EMPLOYEE_ID = '00167'
 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.
SQL> --
SQL> SET OPTIMIZATION LEVEL 'TOTAL TIME';
SQL> select EMPLOYEE_ID, LAST_NAME
cont> from EMPLOYEES
cont> where EMPLOYEE_ID IN ('00167', '00168');
Tables:
  0 = EMPLOYEES
Leaf#01 BgrOnly 0:EMPLOYEES Card=100
  Bool: (0.EMPLOYEE_ID = '00167') OR (0.EMPLOYEE_ID = '00168')
  BgrNdx1 EMPLOYEES_HASH [(1:1)2] Fan=1
    Keys: r0: 0.EMPLOYEE_ID = '00168'
          r1: 0.EMPLOYEE_ID = '00167'
 EMPLOYEE_ID   LAST_NAME
 00167         Kilpatrick
 00168         Nash
2 rows selected
SQL> --
SQL> -- When the SET OPTIMIZATION LEVEL 'DEFAULT' statement
SQL> -- is specified the session will revert to the default FAST FIRST
SQL> -- optimizer tactic.
SQL> --
SQL> SET OPTIMIZATION LEVEL 'DEFAULT';
SQL> select EMPLOYEE_ID, LAST_NAME
cont> from EMPLOYEES
cont> where EMPLOYEE_ID IN ('00167', '00168');
Tables:
  0 = EMPLOYEES
Leaf#01 FFirst 0:EMPLOYEES Card=100
  Bool: (0.EMPLOYEE_ID = '00167') OR (0.EMPLOYEE_ID = '00168')
  BgrNdx1 EMPLOYEES_HASH [(1:1)2] Fan=1
    Keys: r0: 0.EMPLOYEE_ID = '00168'
          r1: 0.EMPLOYEE_ID = '00167'
 EMPLOYEE_ID   LAST_NAME
 00167         Kilpatrick
 00168         Nash
2 rows selected
SQL>
```

Example 2: Using sampled selectivity

This example shows the use of the SET OPTIMIZATION LEVEL command and the resulting use of "Estim" prior to query compile. The estimate (34 rows) is quite close to the final result of 37 rows.

```
SQL> set flags ’strategy,detail,execution’;
SQL> set optimization level ’total time, sampled selectivity’;
SQL> select * from employees where employee_id between ’00000’ and ’00200’;
~Estim EMP_EMPLOYEE_ID Sorted: Split lev=2, Seps=2 Est=34
~Estim EMP_EMPLOYEE_ID Sorted: Split lev=2, Seps=2 Est=34
~S#0005
Tables:
    0 = EMPLOYEES
Leaf#01 BgrOnly 0:EMPLOYEES Card=100
  Bool: (0.EMPLOYEE_ID >= ’00000’ AND (0.EMPLOYEE_ID <= ’00200’)
  BgrNdx1 EMP_EMPLOYEE_ID [1:1] Fan=17
    Keys: (0:EMPLOYEE_ID >= ’00000’) AND 0.EMPLOYEE_ID <= ’00200’)
~Estim EMP_EMPLOYEE_ID Sorted: Split lev=2, Seps=1 Est=17
~E#0005.01(1) Estim   Index/Estimate 1/17
~E#0005.01(1) Bgrndx1 EofData  DBKeys=37  Fetches=0+0  RecsOut=0  #Bufs=30
 EMPLOYEE_ID   LAST_NAME        FIRST_NAME  MIDDLE_INITIAL  ADDRESS_DATA1             ADRESS_DATA_2
   CITY                  STATE   POSTAL_CODE  SEX    BIRTHDAY     STATUS_CODE
 00190         O’Sullivan       Rick        G.              78 Mason Rd.               NULL
   Fremont               NH      03044        M      12-Jan-1923  1
   .
   .
   .
~E#005.01(1) Fin     Buf      DBKeys=37  Fetches=0+32  RecsOut=37
 00174         Myotte           Daniel      V.              95 Princeton Rd.           NULL
   Bennington            MA      03442        M      17-Jan-1948  1

   37 rows selected
SQL>
```

SET QUERY Statement

The SET QUERY statement is used to control query execution within a SQL session.

Environment

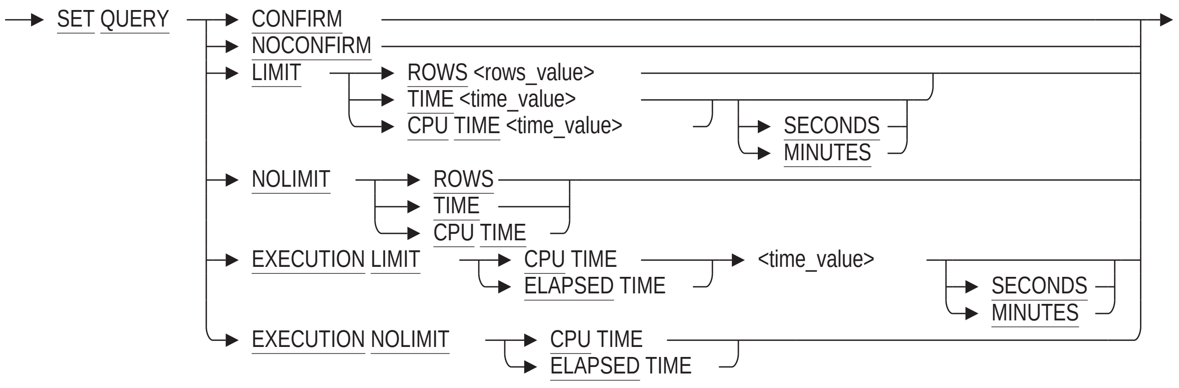
You can use the SET QUERY statement:

- 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

Note that some options for the SET QUERY command may only be used in interactive SQL.

Format

set-query =



Arguments

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.

The SET QUERY CONFIRM statement is only available for interactive SQL.

EXECUTION LIMIT

This option imposes elapsed and CPU time limits on executing queries. This command affects all subsequent queries executed within the Rdb server process. You must be attached to a database to execute this statement. This statement affects all attaches for the current process, not just the current connection.

- CPU TIME time_value [SECONDS | MINUTES]
- ELAPSED TIME time_value [SECONDS | MINUTES]

You can restrict the amount of elapsed time or CPU time used to execute a query. If the query is not complete before the elapsed or CPU time limit is reached, an error message is returned.

The default is unlimited time for the query execution. If you omit the SECONDS and MINUTES keyword then SECONDS is the default. Dynamic SQL options are inherited from the compilation qualifier for the module.

Note

Specifying a query time limit can cause application failure in certain circumstances. For instance, an application that runs successfully during off-peak hours may fail when run during peak hours due to the load on the database.

Use a positive integer for the number of seconds or minutes; 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 a CPU time limit, elapsed time limit and a row limit (using SET QUERY LIMIT), whichever value is reached first stops the query.

Database administraors and application developers can use this feature to prevent users from overloading the system by executing long running, and probably unproductive queries. The database administrator can manage system performance and reduce unnecessary resource usage by setting option limits.

EXECUTION NOLIMIT

This option removes a limit imposed by the SET QUERY EXECUTION LIMIT command.

Use one of the following options.

- ELAPSED TIME
- CPU TIME

EXECUTION NOLIMIT is equivalent to assigning a limit of zero to any of the options using SET QUERY EXECUTION LIMIT.

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 rows_value

  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 defaults are inherited from the compilation qualifier for the module.

- TIME time_value [SECONDS | MINUTES]

  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. If you omit the SECONDS and MINUTES keyword then SECONDS is the default.

  Note

  Specifying a query time limit can cause application failure in certain circumstances. For instance, an application that runs successfully during off-peak hours may fail when run during peak hours due to the load on the database.

- CPU TIME time_value [SECONDS | MINUTES]

  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. If you omit SECONDS and MINUTES keyword then SECONDS is the default. Dynamic SQL options are inherited from the compilation qualifier for the module.

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 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.

NOCONFIRM
Disables the query confirm dialog that was previously enabled using SET QUERY CONFIRM. The SET QUERY NOCONFIRM statement is only available for interactive SQL.

NOLIMIT
This option removes a limit imposed by the SET QUERY LIMIT command.

Use one of the following options.

- ROWS
- TIME
- CPU TIME

NOLIMIT is equivalent to assigning a limit of zero to any of the options using SET QUERY LIMIT.

rows_value
This argument represents the number of rows specified for the SET QUERY argument. It can be a numeric literal, a parameter name (for interactive SQL), or a parameter-marker (for dynamic SQL).

time_value
This argument represents the number of seconds or minutes specified for the SET QUERY statement. It can be a numeric literal, a parameter name (for interactive SQL), or a parameter-market (for dynamic SQL).

Examples

Example 1: Shows the syntax for establishing a row limit within an interactive SQL session.

```
SQL> set query limit rows 10000;
SQL> show query limit;
Query limit Time is OFF
Query limit Row count is 10000 rows
Query limit CPU time is OFF
Execution limit CPU time is OFF
Execution limit Elapsed time is OFF
Execution limit Row count is OFF
SQL> set query nolimit rows;
SQL> show query limit;
Query limit Time is OFF
Query limit Row count is OFF
Query limit CPU time is OFF
Execution limit CPU time is OFF
Execution limit Elapsed time is OFF
Execution limit Row count is OFF
```

Example 2: Uses SET QUERY to establish a two second elapsed time limit for a query, and shows the error message that is displayed.

```
SQL> set query execution limit elapsed time 2 seconds;
SQL> delete from EMPLOYEES;
%RDB-E-EXQUOTA, Oracle Rdb runtime quota exceeded
-RDMS-E-MAXTIMLIM, query governor maximum timeout has been reached
SQL> set query execution nolimit elapsed time;
```

SET QUIET COMMIT Statement

Allows you to control the error reporting behavior when a COMMIT or ROLLBACK statement is executed although there is no active transaction. By default, if there is no active transaction, SQL raises an error when a COMMIT or ROLLBACK statement is executed. If the SET QUIET COMMIT statement is set to ON, then a COMMIT or ROLLBACK statement executes successfully even when there is no active transaction.

Environment

You can use the SET QUIET COMMIT statement:

- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed

Format

```
SET QUIET COMMIT  ──►  on-or-off-value
```

Argument

on-or-off-value
Specifies a string literal or host variable containing the keyword ON or OFF.

The 'ON' argument specifies that if a COMMIT or ROLLBACK statement is executed when there is no active transaction, then SQL will not raise an error. The 'OFF' argument specifies that if a COMMIT or ROLLBACK statement is executed when there is no active transaction, then SQL will raise an error. You can specify the 'ON' and 'OFF' arguments using any case (uppercase, lowercase, or mixed case).

By default, if there is no active transaction, SQL raises an error when the COMMIT or ROLLBACK statement is executed. This default is retained for backward compatibility for applications that want to detect this situation.

Usage Notes

- The following options and qualifiers have the same effect as the SET QUIET COMMIT statement in their respective interfaces:
  - QUIET COMMIT clause for the SQL module language header option
  - /QUIET_COMMIT and /NOQUIET_COMMIT qualifiers for the SQL module language qualifier
  - /SQLOPTIONS=QUIET_COMMIT and /SQLOPTIONS=NOQUIET_COMMIT qualifiers for the SQL language precompiler
- If you issue a COMMIT or ROLLBACK statement within a compound statement, stored procedure, or function, no exception is ever raised when a transaction is not active and you have not issued the SET QUIET COMMIT statement. In effect, the behavior of the SET QUIET COMMIT statement is always on for compound statements, stored procedures, and functions.

- The following dialects, when they are used with the SET DIALECT statement, set the QUIET COMMIT to the ON state: SQL92, SQL99, SQL2011, and ORACLE dialect.

Example

Example 1: Setting the QUIET COMMIT Option On and Off

```
SQL> COMMIT;
%SQL-F-NO_TXNOUT, No transaction outstanding
SQL> SET QUIET COMMIT 'ON';
SQL> ROLLBACK;
SQL> SET QUIET COMMIT 'OFF';
SQL> ROLLBACK;
%SQL-F-NO_TXNOUT, No transaction outstanding
```

SET QUOTING RULES Statement

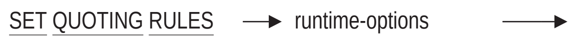
Specifies whether strings within double quotation marks are interpreted as string literals or delimited identifiers in the current connection.

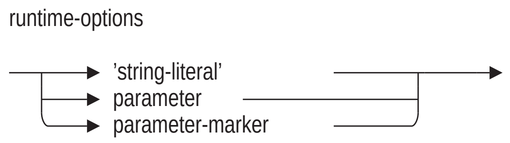
Environment

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





Arguments

′string-literal′
parameter
parameter-marker
Specifies the value of the runtime-options, which must be one of the following:

- SQL89, SQL92, SQL99, SQL2011
- MIA
- ORACLE LEVEL1, ORACLE LEVEL2, ORACLE LEVEL3
- SQLV40

SQLV40
Specifies that SQL interprets strings within double quotation marks as string literals.

The default is SQLV40.

Usage Notes

- The default session setting is SQLV40. This keyword specifies that SQL interprets strings within double quotation marks as string literals. All other dialect keywords specify that SQL interpret 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 one of these options. In addition, you must use one of these options to use multischema database naming.

- 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.
  - SQL99
  - 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 one of these options. 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.

Examples

Example 1: Setting the quoting rules to SQL99

```
SQL> SET QUOTING RULES 'SQL99';
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.

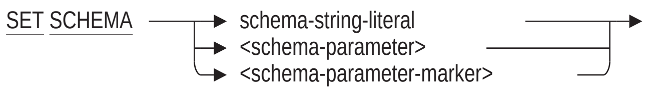
Within one multischema database, tables in different schemas can be used in a single SQL statement; tables in schemas in different databases cannot. If you omit the schema name when you specify an object in a multischema database, SQL uses the default schema name.

Environment

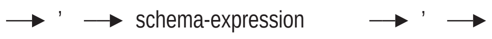
You can use the SET SCHEMA statement:

- 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

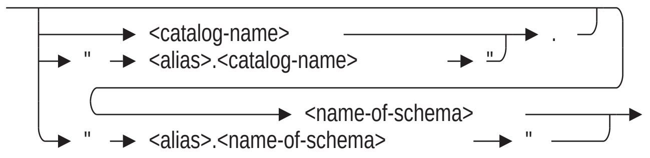
Format



schema-string-literal =



schema-expression =



Arguments

schema-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 uses the user name of the current user.

See Section 2.2.15 for more information on schemas.

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-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.

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 2.2.11. 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 SESSION AUTHORIZATION Statement

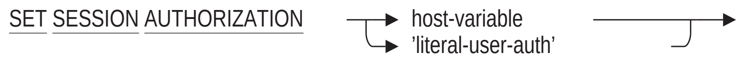
Allows you to transfer the current database attach to another user.

Environment

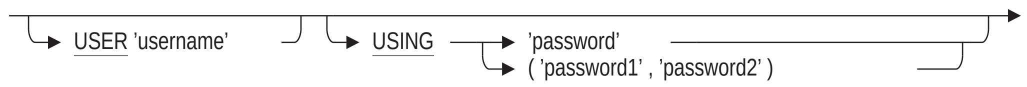
You can use the SET SESSION AUTHORIZATION statement:

- 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



literal-user-auth =



Arguments

host-variable
'literal-user-auth'
Specifies the name of the user and the password to whom the database attach is being transferred as a string literal or a host variable. If a host-variable is specified, it must contain the literal-user-auth as a string literal.

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

USING 'password'
USING ('password', 'password')
A character string literal that specifies the user's password for the user name specified in the USER clause. If the user requires two passwords, then specify both values in parenthesis and separated by a comma.

Usage Notes

- You must have the SELECT privilege on the database to set session authorization.
- The specified user and password (in the USING clause) must be a valid OpenVMS user authorization.
- If the operation is successful, the SESSION_USER and SESSION_UID will be changed to reflect the specified OpenVMS user.

- No transaction can be active when the session authorization is modified by this statement.

Examples

Example 1: Reusing the Current Database Attach for Another User

```
SQL> ATTACH 'FILENAME db$:personnel';
SQL> SET SESSION AUTHORIZATION 'USER ''SMITH'' USING ''SECRET1''';
SQL> SHOW PRIV ON DATABASE RDB$DBHANDLE
Privileges on Alias RDB$DBHANDLE
    (IDENTIFIER =[RDB,SMITH],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
     ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
SQL> SET SESSION AUTHORIZATION 'USER ''JAIN'' USING ''SECRET2''';
SQL> SHOW PRIV ON DATABASE RDB$DBHANDLE
Privileges on Alias RDB$DBHANDLE
    (IDENTIFIER =[RDB,JAIN],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
     ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
```

SET SQLDA Statement

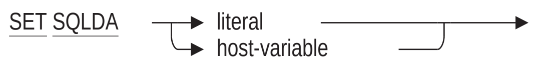
Allows a programmer using Dynamic SQL to alter the way the SQLDA (and SQLDA2) and Dynamic SQL statements are processed by Oracle Rdb.

Environment

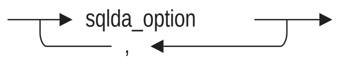
You can use the SET SQLDA statement:

- In Dynamic SQL as a statement to be dynamically executed

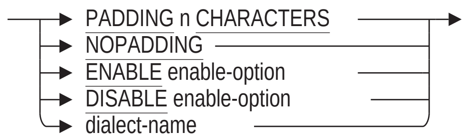
Format



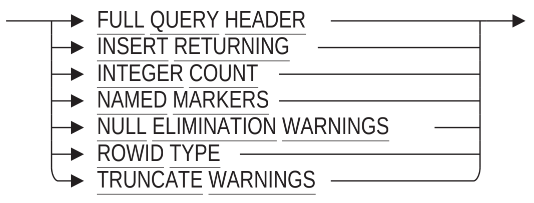
sqlda_options =



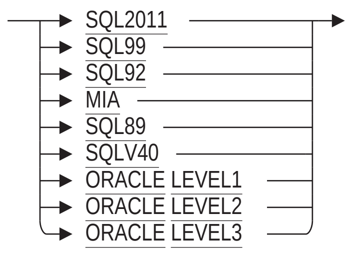
sqlda_option =



enable-option =



dialect-name =



Arguments

Literal
Host-Variable
Parameter passed to the statement. Must be a literal or a host variable containing one or more SQLDA options (see sqlda_options syntax diagram

for details). If more than one option is specified, they must be separated by commas.

sqlda_options
One or more keyword clauses. If more than one clause is specified, they must be separated by commas.

ENABLE
The ENABLE clause activates one of the following behaviors for Dynamic SQL.

- FULL QUERY HEADER - By default, any select expression that is not a column or DBKEY is given an empty SQLNAME in the SQLDA (SQLNAME_LEN is zero). When this option is enabled, an approximation of the select expression is formatted as a label for the expression.

  The SQLNAME_LEN will be between 1 and 62, therefore the expression may be truncated. If any of the SQLDA options ORACLE LEVEL1, ORACLE LEVEL2 or ORACLE LEVEL3 are set, then the SQLNAME_LEN will be limited to 30 as this is the largest name supported by Oracle Database.

  If the dialect is set to any of ORACLE LEVEL1, ORACLE LEVEL2 or ORACLE LEVEL3, then some functions will be presented using Oracle Database names (SYSDATE, SYSTIMESTAMP, ROWID and NVL) instead of the Oracle Rdb SQL names (CURRENT_TIMESTAMP, DBKEY and COALESCE) regardless of the SQL syntax used in the original query.
- INSERT RETURNING - The default behavior of INSERT ... RETURNING when executed by dynamic SQL is to place parameters from the RETURNING INTO clause in to the INPUT SQLDA. This behavior is maintained for backward compatibility. This option allows the programmer to force different (and corrected) behavior for the non-compound use of this statement.

  Note

  If the INSERT RETURNING statement is included in a compound statement then the parameters are handled correctly.

- INTEGER COUNT - The default behavior for Dynamic SQL is to expect the result data type of the COUNT function as BIGINT. When this option is enabled, Dynamic SQL will implicitly cast the result to INTEGER. If this option is disabled, then SQL will revert to a BIGINT result data type.
- NAMED MARKERS - as well as traditional parameters markers (?). Dynamic SQL will now accept named, host-variable style parameter markers. See the Usage Notes for further details and examples.
- NULL ELIMINATION WARNINGS - The default behavior when SET DIALECT establishes the dialect as SQL92, SQL99, SQL2011, or an ORACLE dialect is to generate a warning when an aggregate function (COUNT, MIN, MAX, AVG, STDDEV, etc) eliminates NULL values when computing a result.

  This setting of the SQLDA allows dynamic applications to enable or disable this behavior for all dialects, including SQLV40 (default dialect) and SQL89.
- ROWID TYPE - returns DBKEY values as a special type (SQLDA_ROWID, 455) to make processing of the DBKEY values easier. For instance, in prior releases the SQLDA name field (SQLNAME) for DBKEY entries in the SQLDA was the only way to distinguish these values from other CHAR or VARCHAR columns - it would be either DBKEY or ROWID. If a query

renamed the DBKEY column, then the application had no information in the SQLDA to indicate that the CHAR or VARCHAR value was binary data. In all respects, the SQLDA_ROWID type appears as a fixed length string of octets (possibly containing octets of zero which the C language would treat as a NULL terminator for a string).

- TRUNCATE WARNINGS - The default behavior when SET DIALECT establishes the dialect as SQL92, SQL99, SQL2011, or an ORACLE dialect is to generate an error when an assignment would cause a string value to be truncated. This setting of the SQLDA allows dynamic applications to enable or disable this behavior for all dialects, including SQLV40 (default dialect) and SQL89.

Note

Trailing spaces characters are ignored when determining if a string is truncated.

DISABLE
The DISABLE clause deactivates one of the specified behaviors for Dynamic SQL. See ENABLE clause for a list of options.

ORACLE LEVEL1
ORACLE LEVEL2
ORACLE LEVEL3
Any of these options will set the SQLDA to supply enhanced semantics. These options are currently reserved for the use of the OCI Services for Rdb product that is part of Oracle Rdb SQL/Services component. This setting also implicitly enables NAMED MARKERS.

PADDING n CHARACTERS
This option directs SQL to configure the SQLDA with larger CHARACTER VARYING strings than would normally be seen. The value of n is an unsigned numeric literal that specifies the number of characters that are added to the estimated length. Any CHARACTER (CHAR) types are converted to CHARACTER VARYING (VARCHAR). This rule is applied to comparison operators <, <=, >, >=, =, <>, and string functions (STARTING WITH, CONTAINING).

NOPADDING
This option sets the number of padding characters to 0. This also implies that derived CHARACTER (CHAR) types are not converted to CHARACTER VARYING (VARCHAR) when PADDING CHARACTERS is used. This is the default setting.

Note

Oracle recommends that applications always check for SQLDA_CHAR and SQLDA_VARCHAR so that the correctly formatted data is made available to SQL.

SQL2011
SQL99
SQL92
MIA
SQL89
SQLV40
Any of these options will revert to the default semantic for the SQLDA which includes disabling NAMED MARKERS.

Usage Notes

- The ORACLE LEVEL1, ORACLE LEVEL2 and ORACLE LEVEL3 settings are reserved for use by Oracle Corporation. Current behavior of this setting may change with any given release based on requirements of the OCI Services for Rdb component. This setting changes the usage of various SQLDA and SQLDA2 fields.
- Keywords may not be abbreviated and the clauses must be fully specified.
- The SET DIALECT command will implicitly enable NAMED MARKERS if the dialect is changed to one of the Oracle dialects.
- The SET DIALECT command will implicitly disable NAMED MARKERS if the dialect is changed to any dialect other than an ORACLE dialect.
- When NAMED MARKERS are enabled, the contents of the SQLDA and SQLDA2 will reflect one entry for each name. When traditional parameter markers are used, a SQLDA (or SQLDA2) entry will exist for each marker (?) encountered. This change in behavior can simplify the query encoding as well lead to more efficient strategy creation.

Example

Example 1: Using the NAMED MARKERS feature

This example shows that enabling the NAMED MARKERS feature will allow SQL to prompt for one value and the displayed Rdb strategy shows that only one variable is used.

```
-> SET SQLDA 'ENABLE NAMED MARKERS';
-> SELECT LAST_NAME FROM EMPLOYEES WHERE FIRST_NAME = :F_NAME AND LAST_NAME <>
:F_NAME;
in:  [0] typ=449 len=46
out: [0] typ=453 len=14
[SQLDA - reading 1 fields]
-> Alvin
Tables:
  0 = EMPLOYEES
Conjunct: (0.FIRST_NAME = <var0>) AND (0.LAST_NAME <> <var0>)
Get     Retrieval sequentially of relation 0:EMPLOYEES
 0/FIRST_NAME/Varchar(42/46): Alvin
[SQLDA - displaying 1 fields]
 0/LAST_NAME: Toliver
[SQLDA - displaying 1 fields]
 0/LAST_NAME: Dement
```

Example 2: Using the PADDING feature

The following example shows that the derived type for the named parameter MI is a SQLDA_CHAR (453) of length 1. The input data ('AA') is truncated on assignment and the incorrect results are returned. By adding a small padding the type is changed to SQLDA_VARCHAR (449) of length 3 and a correct comparison is performed.

```
-> ATTACH 'filename sql$database';
-> SET SQLDA 'enable named markers, nopadding';
-> SELECT LAST_NAME FROM EMPLOYEES WHERE MIDDLE_INITIAL = :MI;
in:  [0] typ=453 len=1
out: [0] typ=449 len=18
[SQLDA - reading 1 fields]
-> AA
[SQLDA - displaying 1 fields]
 0/LAST_NAME: Toliver
[SQLDA - displaying 1 fields]
 0/LAST_NAME: Lengyel
[SQLDA - displaying 1 fields]
 0/LAST_NAME: Robinson
[SQLDA - displaying 1 fields]
 0/LAST_NAME: Ames
-> SET SQLDA 'padding 2 characters';
-> SELECT LAST_NAME FROM EMPLOYEES WHERE MIDDLE_INITIAL = :MI;
in:  [0] typ=449 len=7
out: [0] typ=449 len=18
[SQLDA - reading 1 fields]
-> AA
-> EXIT;
Enter statement:
```

Note that the VARCHAR requires an extra 4 bytes for the length information in the SQLDA2 used by the Dynamic SQL testing program.

Example 3: Using the ENABLE INTEGER COUNT feature

The following example uses Dynamic SQL and accepts various statements from the user. When using SET SQLDA, the returned data type of COUNT is altered from the default (BIGINT) to INTEGER.

```
-> ATTACH 'filename sql$database';
inputs: 0
-> SELECT COUNT(*) FROM RDB$DATABASE;
inputs: 0
out: [0] typ=Bigint {505} len=8
[SQLDA - displaying 1 fields]
 0/: 1
-> SET SQLDA 'enable integer count';
inputs: 0
-> SELECT COUNT(*) FROM RDB$DATABASE;
inputs: 0
out: [0] typ=Integer {497} len=4
[SQLDA - displaying 1 fields]
 0/: 1
-> EXIT;
```

Example 4: Enabling FULL QUERY HEADER

The following example uses Dynamic SQL and accepts various statements. The tool displays the label from the SQLDA as a description for the user.

```
Enter statement:
attach 'filename sql$database';
inputs: 0
Enter statement:
set sqlda 'enable full query header';
inputs: 0
Enter statement:
select employee_id, first_name || last_name, extract(year from birthday)
from employees
where employee_id = '00164';
inputs: 0
out: [0] typ=Char {453} len=5
out: [1] typ=Char {453} len=24
out: [2] typ=Integer {497} len=4
[SQLDA - displaying 3 fields]
 0/EMPLOYEE_ID: 00164
 1/CONCAT(FIRST_NAME,...): Alvin     Toliver
 2/EXTRACT(YEAR FROM BIRTHDAY): 1947
Enter statement:
```

Example 5: Using the ENABLE TRUNCATE WARNINGS feature

The following example uses Dynamic SQL to execute various INSERT statements. The tool displays the error reported for string truncation.

```
-> CREATE TABLE SAMPLE_TABLE (COL1 CHAR);
-> INSERT INTO SAMPLE_TABLE VALUES ('xxx');
inputs: 0
-> !;
-> SET SQLDA 'enable truncate warnings';
inputs: 0
-> INSERT INTO SAMPLE_TABLE VALUES ('xxx');
inputs: 0
Error -306:
%RDB-E-TRUN_STORE, string truncated during assignment to a column
-> !;
-> SET SQLDA 'disable truncate warnings';
inputs: 0
-> INSERT INTO SAMPLE_TABLE VALUES ('xxx');
inputs: 0
```

Example 6: Using the NULL ELIMINATION WARNINGS feature

The following example uses Dynamic SQL to execute a COUNT function on a column that has some values set to NULL. The tool displays the warning reported in such cases.

```
Enter statement:
SET DIALECT 'SQL99';
inputs: 0
Enter statement:
SELECT COUNT (MIDDLE_INITIAL) FROM EMPLOYEES;
inputs: 0
out: [0] typ=Bigint {505} len=8
--> reported warning; sqlcode=1003
[SQLDA - displaying 1 fields]
 0/: 64
Enter statement:
SET SQLDA 'DISABLE NULL ELIMINATION WARNINGS';
inputs: 0
Enter statement:
SELECT COUNT (MIDDLE_INITIAL) FROM EMPLOYEES;
inputs: 0
out: [0] typ=Bigint {505} len=8
[SQLDA - displaying 1 fields]
 0/: 64
Enter statement:
```

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.

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)
- Horizontal partition specification (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)

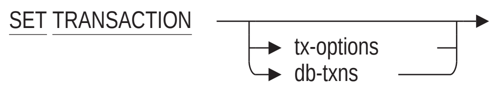
The Arguments section explains these options in more detail.

Environment

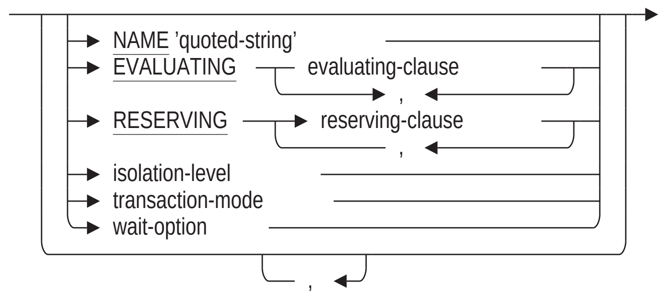
You can use the SET TRANSACTION statement:

- 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

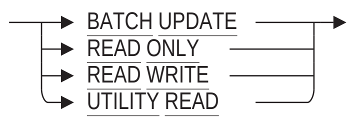
Format



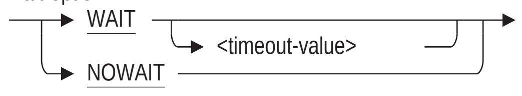
tx-options =



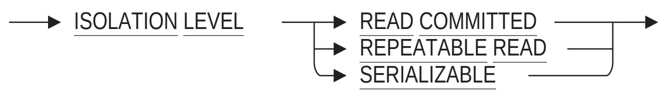
transaction-mode =



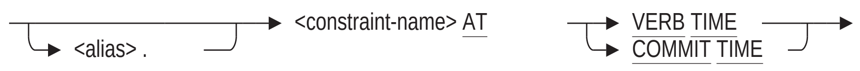
wait-option =



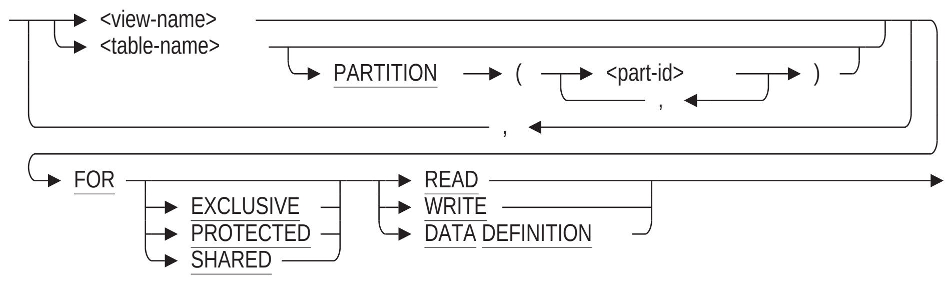
isolation-level =



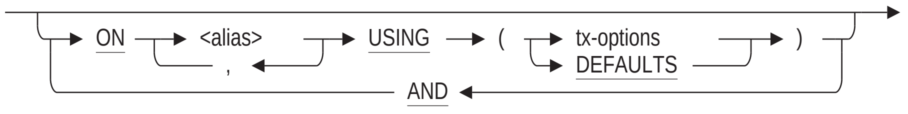
evaluating-clause =



reserving-clause =



db-txns =



Arguments

alias
Specifies the alias for a constraint. See the Usage Notes for information on using aliases for a multischema database.

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 Rdb Guide to SQL Programming*.

The batch-update transaction permits updates to the database without creating a recovery-unit journal (.ruj) file. Therefore, any rows or indices modified during the transaction cannot be rolled back because Oracle Rdb does not maintain before-images of the changed records.

For example, if you need a large test database for development purposes, a batch-update transaction loads the database but bypasses the journaling facilities. If the load fails, you must create the database again.

Because you cannot use batch-update transactions with distributed transactions, you should define the SQL$DISABLE_CONTEXT logical name as “True” before you start a batch-update transaction. (Distributed transactions require that you are able to roll back 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 correcting 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.

constraint-name
Specifies the name of a constraint.

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.

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 ignored.

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 8–6 describes the different share modes.

Table 8–6 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 during query processing.

Table 8–7 compares the effect of different lock specifications.

Table 8–7 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

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 8–8 shows the phenomena permitted for the isolation levels that you can explicitly specify with the SET TRANSACTION statement.

Table 8–8 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 Rdb Guide to SQL Programming* for further information about specifying isolation levels in transactions.

NAME transaction-name
Supplies a title for the transaction. This information is displayed by the SET FLAGS TRANSACTION keyword.

```
SQL> declare transaction read write name 'default-transaction';
SQL> select * from rdb$database;
~T Compile transaction (3) on db: 1
~T Transaction Parameter Block: (len=23)
0000 (00000) TPB$K_VERSION = 1
0001 (00001) TPB$K_BUFFER_NAME "default-transaction"
0016 (00022) TPB$K_WRITE (read write)
~T Start_transaction (3) on db: 1, db count=1
   .
   .
   .
```

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.

PARTITION (part-id, ...)

When used with the RESERVING clause, specifies a list of partitions so that only a subset of the table's partitions are reserved. For example, an application could submit several processing jobs that each reserved a separate partition of the table for EXCLUSIVE access. The default, if this clause is omitted, is to reserve all partitions with the specified mode. An error is reported if the application references a partition of the table that was not reserved.

part-id

The partition can be identified as a number that corresponds to the ordinal position of the partition or the partition name. The name is either defined by the PARTITION clause of the CREATE STORAGE MAP statement or one that is system generated. See the output from SHOW STORAGE MAP which lists both the partition names and the ordinal position.

```
SQL> show storage map employees_map
     EMPLOYEES_MAP
 For Table: EMPLOYEES
 Placement Via Index: EMPLOYEES_HASH
 Partitioning is:   UPDATABLE

 Partition information for storage map:
 Compression is:    ENABLED
  Partition: (1) SYS_P00079
   Storage Area: EMPIDS_LOW
  Partition: (2) SYS_P00080
   Storage Area: EMPIDS_MID
  Partition: (3) SYS_P00081
   Storage Area: EMPIDS_OVER
```

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 or alter metadata at the same time as other users on the same table. This clause can be used only in read/write transactions. See the Usage Notes for additional information.

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 are implicitly isolation level serializable.

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 to maintain isolation level serializable.
- If your query uses indexes, SQL may lock part of an index, which has the effect of locking several rows.

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.

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 starting a transaction, there are three different values that are used to determine the lock timeout interval for that transaction. Those values are:

1. The value specified in the SET TRANSACTION statement
2. The value stored in the database as specified in CREATE or ALTER DATABASE
3. The value of the logical name RDM$BIND_LOCK_TIMEOUT_INTERVAL

The timeout interval for a transaction is the smaller of the value specified in the SET TRANSACTION statement and the value specified in CREATE DATABASE. However, if the logical name RDM$BIND_LOCK_TIMEOUT_INTERVAL is defined, the value of this logical name overrides the value specified in CREATE DATABASE.

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.

UTILITY READ
This is a special type of READ ONLY transaction performed by utilities that wish to read the system tables and incur as little overhead as possible. Such utilities include RMU, Interactive SQL, Dynamic SQL and OCI Services for Rdb. See the Usage Notes for more detail.

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. 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.

Table 8–9 compares the effects of different lock specifications on multiuser access.

Table 8–9 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 EXCLUSIVE WRITE EXCLUSIVE DATA DEFINITION	No access	No one else can use the table.	No effect.
PROTECTED READ	PROTECTED READ 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 PROTECTED WRITE SHARED READ 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.
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 create and alter metadata for the table 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.

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 8–10 summarizes the defaults for each option and combination of options.

Table 8–10 Defaults for the SET and DECLARE TRANSACTION Statements

Option	Default
Transaction Mode: • READ WRITE • READ ONLY	The default is READ WRITE. Which transaction mode, if any, you specify determines the default lock specification.
Lock Specification:	
• RESERVING	• 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. • If you specify a read/write transaction and include a RESERVING clause, the default is SHARED. • If you do not specify a transaction mode but do include a RESERVING clause, the default is SHARED. • If you specify a read-only transaction, the default is SHARED READ, whether or not you specify a RESERVING clause.
Share Mode: • SHARED • PROTECTED • EXCLUSIVE	The default is SHARED.

(continued on next page)

Table 8–10 (Cont.) Defaults for the SET and DECLARE TRANSACTION Statements

Option	Default
Lock Type: • READ • WRITE • DATA DEFINITION	• If you specify a read/write transaction, the default is WRITE. • If you specify a read-only transaction, the default, and only allowed lock type, is READ.
Concurrency Option: • ISOLATION LEVEL READ COMMITTED • ISOLATION LEVEL REPEATABLE READ • ISOLATION LEVEL SERIALIZABLE	The default is ISOLATION LEVEL SERIALIZABLE.
Wait Mode: • WAIT • NOWAIT	The default is WAIT.

Usage Notes

- For each database specified the following restrictions exist:
  - Only one of the clauses READ ONLY, READ WRITE or BATCH UPDATE may be used.
  - No other clauses may be specified with BATCH UPDATE.
  - Only one of the clauses WAIT and NOWAIT may be used.
  - ISOLATION LEVEL may only be specified once.
- The clauses can be specified in any order.
- The quoted-string provided for NAME can be up to 255 octets in length.
- You cannot use the SET TRANSACTION statement in an ATOMIC compound statement.
- The SET TRANSACTION statement may not be executed from a SQL function or trigger or any stored porcedure called from a SQL function or trigger.
- 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 unless they use the SHARED DATA DEFINITION clause.
- 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.
- 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.
- 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
```

- 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 Rdb 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.

- Explicitly calling the distributed transaction manager lets you control when your application transactions across several 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 DEADLOCK after each database operation. In addition, your program should check for 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 2.2.11.
- 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. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information about this logical name 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 Rdb 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, AUTOMATIC, or DEFAULT expression is implicitly reserved in SHARED READ mode by the referencing statement. If the table is indirectly accessed by a stored function then use LOCK TABLE to reserve the table.
- The partition clause is not permitted if a table does not have a storage map, or has a vertically partitioned storage map (that is, it uses the STORE COLUMNS clause). If an index and the storage map have identical STORE clauses, then both are locked using the same list of partition numbers.
- Using the PARTITION clause requires careful database and application design. If the indexes are partitioned using different partitioning keys or different value ranges, then cross-partition updates might lead to deadlocks and other lock conflicts between concurrent update processes.
- The PARTITION clause accepts a list of partition names or a list of partition ordinal values. You may not mix numeric and named notations.
- Duplicate partition names in the RESERVING clause will cause an exception. Review the RESERVING clause and correct the partition names.

  ```
  set transaction
      read write
      evaluating
          job_history_foreign1 at verb time
          ,salary_history_foreign1 at verb time
      reserving
          employees partition (SYS_P00080, SYS_P00080) for exclusive write
  ;
  %RDB-E-BAD_TPB_CONTENT, invalid transaction parameters in the transaction
  parameter block (TPB)
  -RDMS-E-DUPPARTNAME, partition SYS_P00080 for table EMPLOYEES already used
  ```

- Unknown partition names in the RESERVING clause (which might occur due to a change in the storage map definition) will cause an exception. Use the SHOW STORAGE MAP statement to review the partition names.

```
create module mod_testing1a
 language sql
 procedure proc_xa ();
 begin not atomic
 set transaction
    read write
    evaluating salary_history_foreign1 at verb time
    reserving employees partition (SYS_P00080, "UNKNOWN", SYS_P00081)
              ,departments for protected write;
  commit;
 end;
end module;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDB-E-BAD_TPB_CONTENT, invalid transaction parameters in the transaction
parameter block (TPB)
-RDMS-F-PARTNEXTS, partition "UNKNOWN" does not exist in this map or index
"EMPLOYEES_MAP"
```

- By default, a transaction that reserves a table for EXCLUSIVE access does not reserve the LIST (segmented string) area for exclusive access. Because the LIST area is usually shared by many tables, SHARED access is assumed by default to permit updates to the other tables.

  This means that when you run an import operation or when an application updates a table reserved for EXCLUSIVE access, you might notice that the snapshot storage area (.snp) grows. This is because of the I/O to the LIST area that is performed by default when SHARED WRITE mode is in use.

  However, if you attach to the database using an SQL ATTACH or IMPORT statement and you specify the RESTRICTED ACCESS clause, then all storage areas are accessed in EXCLUSIVE mode. Use this clause to eliminate the snapshot I/O and related overhead if you are performing a lot of I/O to the LIST storage areas (for example, when you are restructuring the database, or dropping a large table containing LIST OF BYTE VARYING columns and data).

- UTILITY READ directs Rdb to choose the best transaction mode based on various database and user preferences. Ideally, Rdb would start a READ ONLY transaction but may need to upgrade this to a different type of transaction based on the environment.

  1. If SNAPSHOTS are ENABLED DEFERRED, then a READ ONLY transaction will be stalled until such time as current READ WRITE transactions commit and then new READ WRITE transactions will start writing SNAPSHOT rows. Utility read will try to avoid this overhead by using a READ WRITE transaction.
  2. Rdb uses the TRANSACTION MODES which were defined by CREATE or ALTER DATABASE, plus those that might be inherited from the current user's PROFILE (see the CREATE and ALTER PROFILE statement). It might be that READ ONLY transactions are forbidden or that some types of READ WRITE transactions are not possible.
  3. If this is a standby database (see the HOT STANDBY feature), then only READ ONLY transactions are permitted.
  4. If Rdb is forced to use a READ WRITE transaction, it will implicitly use ISOLATION LEVEL READ COMMITTED and also disable the ability to perform DELETE, INSERT and UPDATE operations.

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 a 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.

Read-only transactions use a snapshot version of the data, and therefore you might encounter older values in the data your application retrieves. because another transaction using a read/write transaction might be updating a table.

The snapshot file represents a before-image of the database rows that the other program is updating. If you require The very latest data, you should specify read/write access for both databases, and permit other users to read one of the databases by including the shared read 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 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 (LAST_NAME);
SQL> --
SQL> -- Commit the transaction immediately.
SQL> --
SQL> COMMIT;
```

Example 8: Reserving a Partition

```
SQL> -- This example locks only the second partition of
SQL> -- the EMPLOYEES table in exclusive write mode.
SQL> -- The advantage of this is that the process can insert,
SQL> --  update, or delete from this partition without writing
SQL> -- to the snapshot (.snp) file, and in general, uses fewer
SQL> --  resources for operations on the partition.
SQL> SET TRANSACTION READ WRITE
cont> RESERVING EMPLOYEES PARTITION (2) FOR EXCLUSIVE WRITE;
```

Example 9: Interaction between RESERVING clause and column DEFAULT values

This example examines the interaction between the RESERVING clause and DEFAULT values that reference tables (either directly and indirectly). The RESERVING clause of SET TRANSACTION limits the transaction to just those tables listed for the transaction.

Tables directly referenced by constraints, triggers, COMPUTED BY columns, AUTOMATIC columns and DEFAULT values are implicitly reserved for SHARED READ. However, if these definitions reference the table indirectly via a stored function then that table is not considered for automatic reservation.

This example uses DEFAULT value to contrast three different mechanisms and their interactions with the RESERVING clause. The same technique could be applied to other definitions such as triggers and constraints.

The DEFAULT value is derived from a secondary table (DEFAULTS) that holds one value for each valid user of the database. The DEFAULT is retrieved based on the value of CURRENT_USER. In the three tables below the value is either directly fetched (SAMPLE_TABLE2), or via a stored function (SAMPLE_TABLE1, and SAMPLE_TABLE3).

The SQL function GET_DEFAULT3 includes a LOCK TABLE statement to ensure that the table is correctly reserved. Oracle recommends this approach since it relieves the programmer from knowing which tables might be required when coding a RESERVING clause for a transaction.

```
SQL> set dialect 'sql99';
SQL>
SQL> create table DEFAULTS
cont>     (user_id      rdb$object_name primary key,
cont>      valid_number integer);
SQL> insert into DEFAULTS values ('SMITH', 100);
1 row inserted
SQL>
SQL> create module UTL1
cont>     function GET_DEFAULT1 ()
cont>     returns integer
cont>     not deterministic;
cont>     return (select valid_number from DEFAULTS
cont>             where user_id = CURRENT_USER);
cont> end module;
SQL>
SQL> create table SAMPLE_TABLE1
cont>     (id         integer identity,
cont>      quantity   integer
cont>                 default GET_DEFAULT1 ()
cont>     );
SQL>
SQL> create table SAMPLE_TABLE2
cont>     (id         integer identity,
cont>      quantity   integer
cont>                 default (select valid_number from DEFAULTS
cont>                          where user_id = CURRENT_USER)
cont>     );
SQL>
SQL> create module UTL3
cont>     function GET_DEFAULT3 ()
cont>     returns integer
cont>     not deterministic;
cont>     begin
cont>     lock table DEFAULTS for shared read mode;
cont>     return (select valid_number from DEFAULTS
cont>             where user_id = CURRENT_USER);
cont>     end;
cont> end module;
SQL>
SQL> create table SAMPLE_TABLE3
cont>     (id         integer identity,
cont>      quantity   integer
cont>                 default GET_DEFAULT3 ()
cont>     );
SQL>
SQL> commit;
```

The following transactions succeed or fail as explained in the example.

```
SQL> /*
***> Fails because the module references a table that is not reserved
***> */
SQL> set transaction read write
cont>     reserving SAMPLE_TABLE1 for shared write;
SQL> insert into SAMPLE_TABLE1 default values;
%RDB-E-UNRES_REL, relation DEFAULTS in specified request is not a
relation reserved in specified transaction
SQL> rollback;
SQL>
SQL> /*
***> Succeeds because direct access to the table from the DEFAULT
***> is implicitly added to the reserving list as SHARED READ
***> */
SQL> set transaction read write
cont>     reserving SAMPLE_TABLE2 for shared write;
SQL> insert into SAMPLE_TABLE2 default values;
1 row inserted
SQL> rollback;
SQL>
SQL> /*
***> Succeeds because the routine adds the table to the reserved
***> table list using LOCK TABLE.
***> */
SQL> set transaction read write
cont>     reserving SAMPLE_TABLE3 for shared write;
SQL> insert into SAMPLE_TABLE3 default values;
1 row inserted
SQL> rollback;
SQL>
```

SET VIEW UPDATE RULES Statement

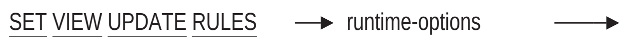
Specifies whether or not SQL applies the ANSI/ISO SQL standard for updatable views to views created during a session.

Environment

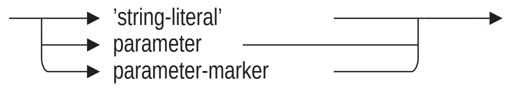
You can use the SET VIEW UPDATE RULES statement:

- 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

'string-literal'
parameter
parameter-marker
Specifies the value of runtime-options, which must be one of the following:

- SQL89, SQL92, SQL99, SQL2011
- MIA
- ORACLE LEVEL1, ORACLE LEVEL2, ORACLE LEVEL3
- SQLV40

Usage Notes

- The dialect keyword 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 session default is SQLV40.

- All other dialect keywords specify that the ANSI/ISO SQL standard for updatable views is applied to all views created during the session. 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.
- 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 SQL99

```
SQL> ATTACH 'ALIAS ENV1 FILENAME ENVIRONMENT';
SQL> CONNECT TO 'ALIAS ENV1 FILENAME ENVIRONMENT' AS 'TEST';
SQL> SHOW CONNECTIONS TEST
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
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
Display character set is UNSPECIFIED
```

```
Alias ENV1:
        Identifier character set is DEC_MCS
        Default character set is DEC_MCS
        National character set is KANJI
SQL> --
SQL> -- Change the environment for view rules from SQLV40 to SQL99
SQL> --
SQL> SET VIEW UPDATE RULES 'SQL99';
SQL> SHOW CONNECTIONS TEST
Connection: TEST
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
Dialect: SQLV40
Default character unit: OCTETS
Keyword Rules: SQLV40
View Rules: ANSI/ISO
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
Display character set is UNSPECIFIED

Alias ENV1:
        Identifier character set is DEC_MCS
        Default character set is DEC_MCS
        National character set is KANJI
```

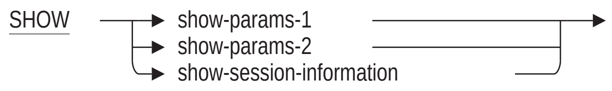
SHOW Statement

Displays information about database entities and information about the interactive SQL session.

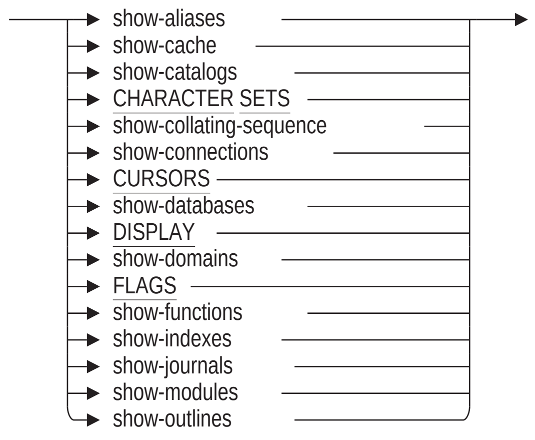
Environment

You can use the SHOW statement only in interactive SQL.

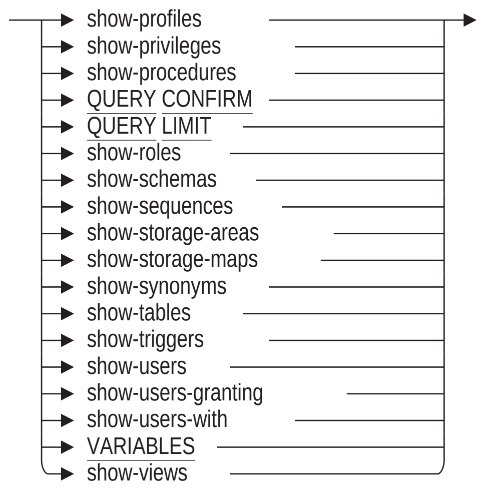
Format



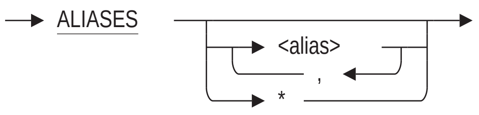
show-params-1 =



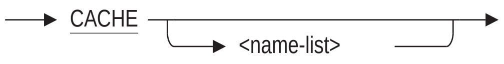
show-params-2 =



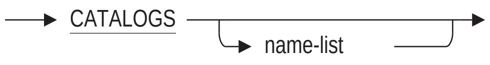
show-aliases =



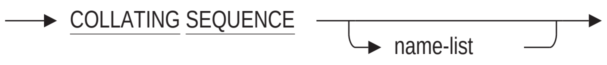
show-cache =



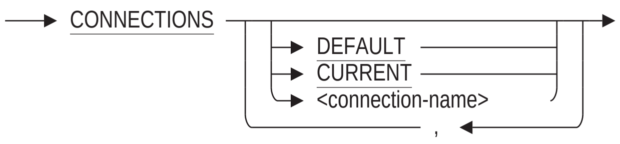
show-catalogs =



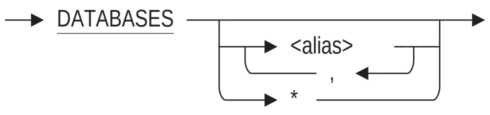
show-collating-sequence =



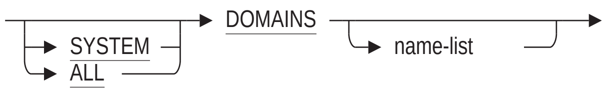
show-connections =



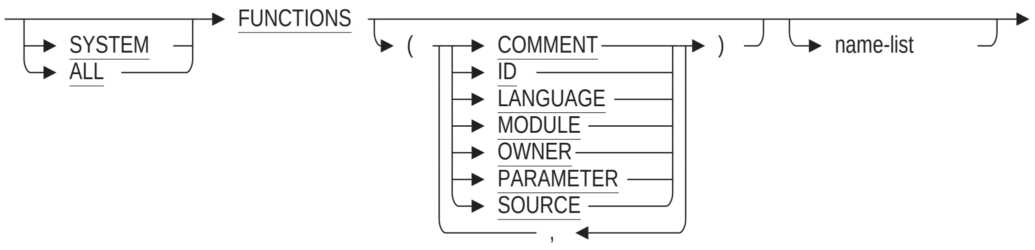
show-databases =



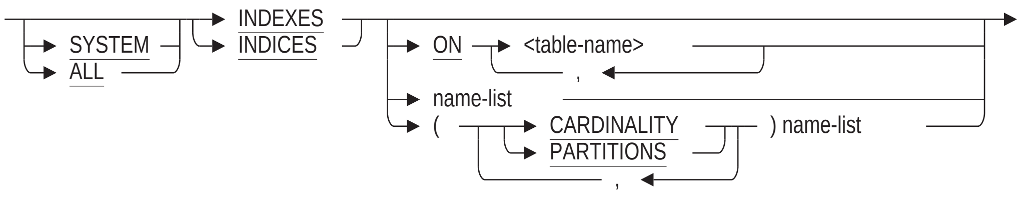
show-domains =



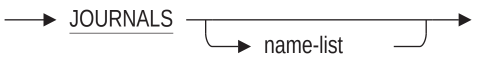
show-functions =



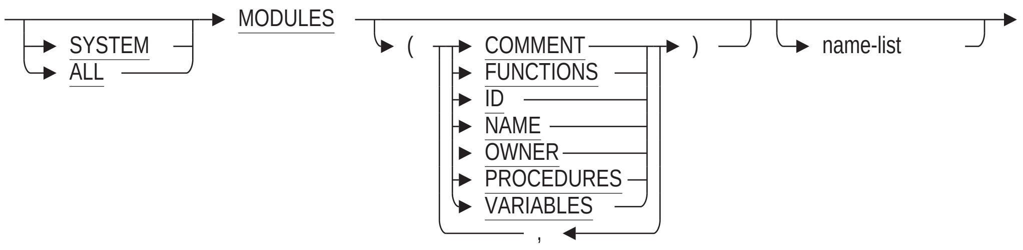
show-indexes =



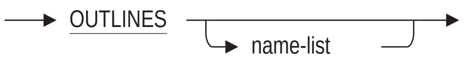
show-journals



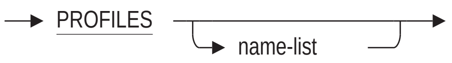
show-modules =



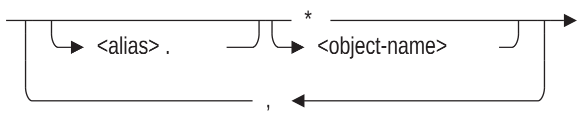
show-outlines



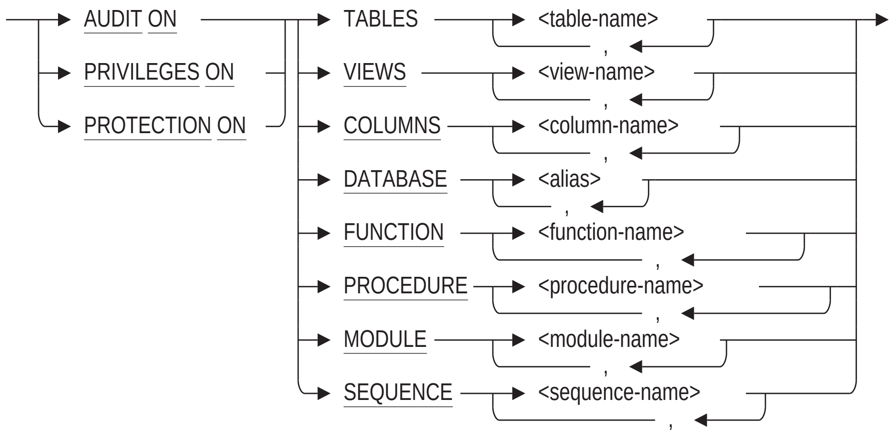
show-profiles =



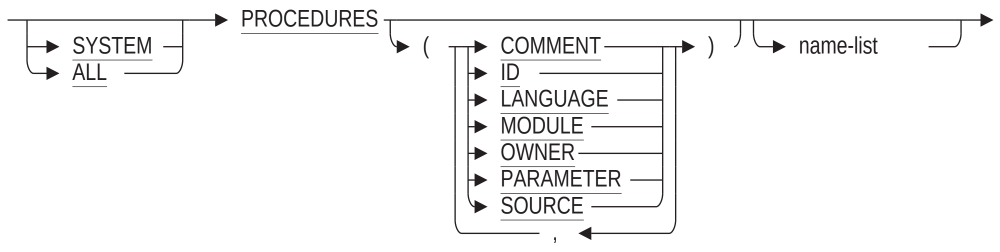
name-list =



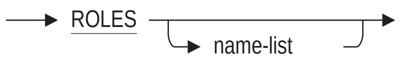
show-privileges =



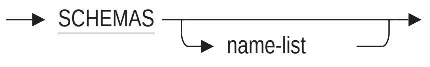
show-procedures =



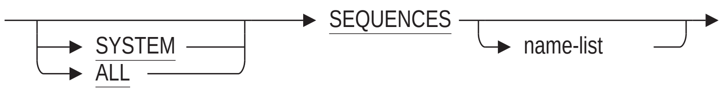
show-roles =



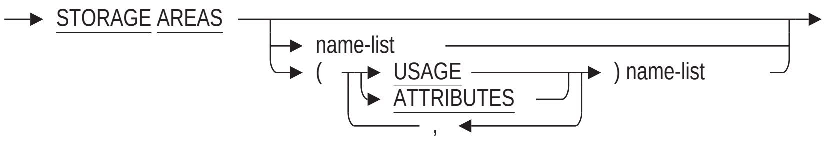
show-schemas =



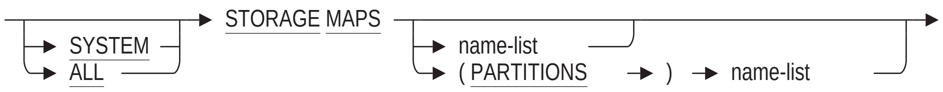
show-sequences=



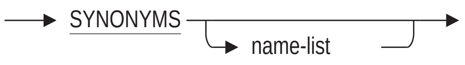
show-storage-areas =



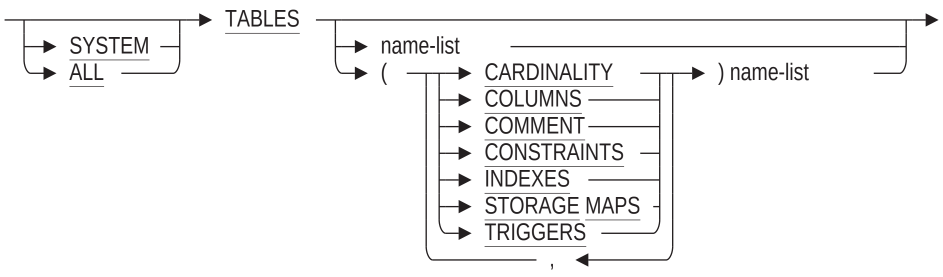
show-storage-maps =



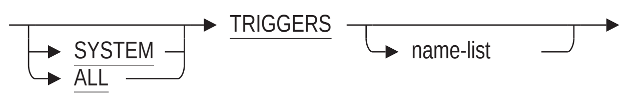
show-synonyms =



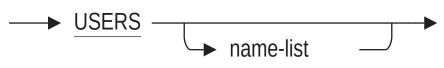
show-tables =



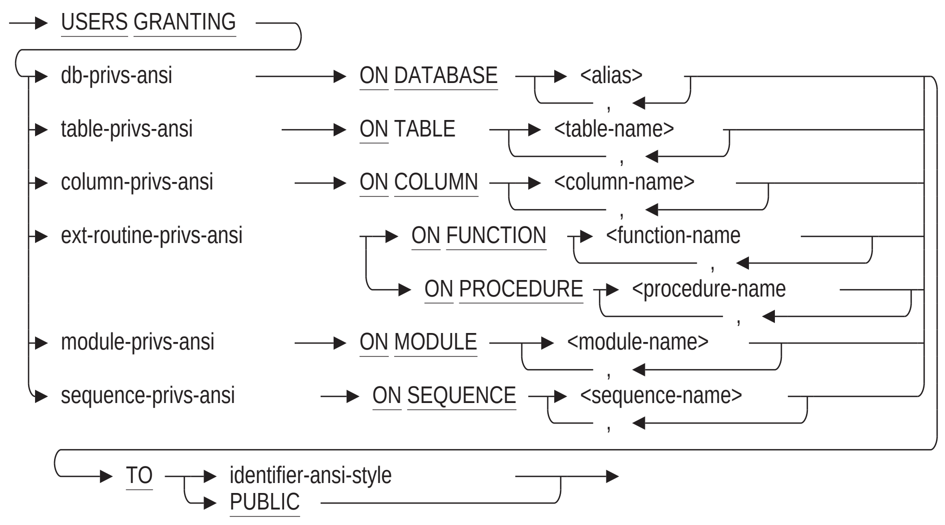
show-triggers =



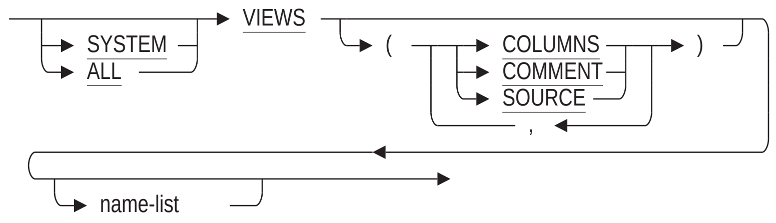
show-users =



show-users-granting =



show-views =



db-privs-ansi =

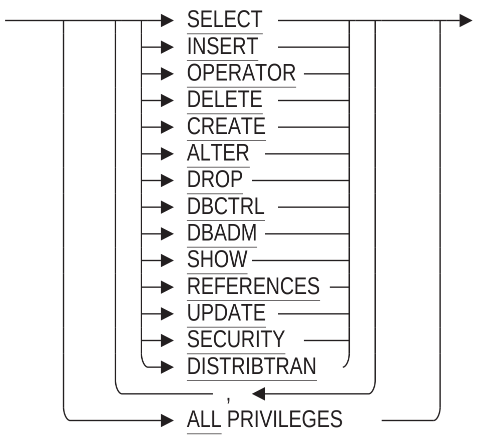
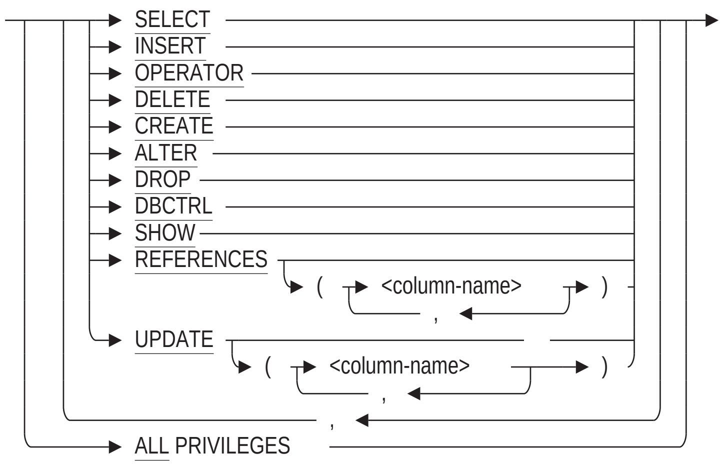
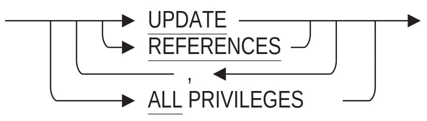


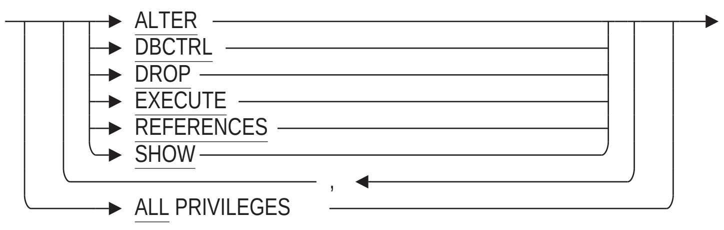
table-privs-ansi =



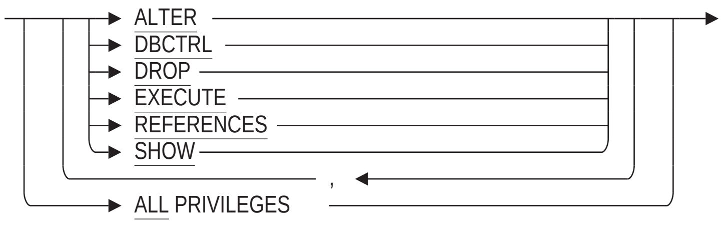
column-privs-ansi =



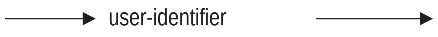
ext-routine-privs-ansi =



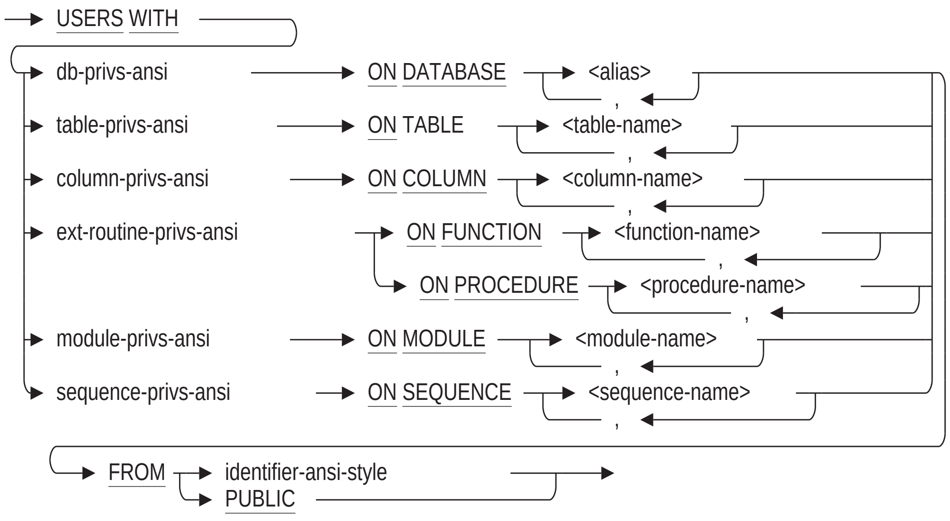
module-privs-ansi =



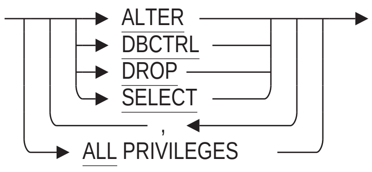
identifier-ansi-style =



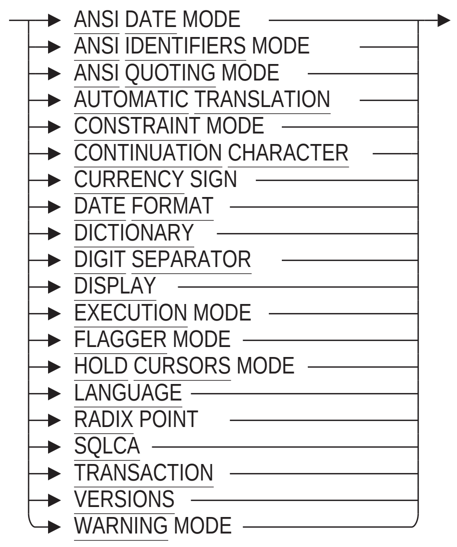
show-users-with =



sequence-privs-ansi =



show-session-information =



Arguments

*****
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 objects in the default database. If you precede the wildcard with an alias, SQL displays information about objects in that database.

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.

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 ANSI/ISO. By default, both data types are interpreted as OpenVMS format.

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 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 Appendix 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.

Use the SET QUOTING RULES statement to change the quoting rules.

AUDIT ON

The SHOW AUDIT statement displays all of the audit information for the specified databases, tables, views, columns, external functions, external procedures, modules, or sequences.

When performing SHOW ON DATABASE, the list of audit identifiers will be listed. For other database objects, only the privileges specified to cause auditing are listed.

AUTOMATIC TRANSLATION

Displays the current setting as established using SET AUTOMATIC TRANSLATION.

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.

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.

CONTINUE CHARACTER
Displays the value for the continuation character, as established using SET CONTINUE CHARACTER.

CURRENCY SIGN
Displays the currency indicator, such as the dollar sign ($), that will be used in output displays.

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.

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.

DATE FORMAT
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.

DICTIONARY
Displays the current default dictionary directory in the data dictionary.

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.)

DISPLAY
Displays the current settings as established using SET DISPLAY, SET FEEDBACK, SET HEADING, SET LINE LENGTH (or SET LINESIZE), SET PAGE LENGTH (or SET PAGESIZE), SET TIMING and SET NULL. Some values (such as line and page length) are determined from the OpenVMS terminal characteristics when starting interactive SQL.

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.

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 statement in your session, SQL will not execute subsequent statements.

You can use the SET NOEXECUTE 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.

FLAGS
Displays the database system debug flags that are enabled for the current session.

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.

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:
COMMENT	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.
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.

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.

You Specify This:	SQL Performs This Action:
CARDINALITY	Adds the index and column prefix cardinality values to the SHOW output.
PARTITIONS	Displays the index partitions showing the partition name and number the name of the storage area used for the partition.

JOURNALS
Displays information about specified journal files. SQL displays the name of the file specification and, if created, the backup file specification.

LANGUAGE
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.

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:
COMMENT	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 a definer's rights module, the definer's user name displays, otherwise for an invoker's rights module the output will be blank.
PROCEDURES	The stored procedures contained in the module.
VARIABLES	Displays module global variables.

name-list
Most SHOW statements accept an optional name-list which can specify the name of the object, or a wildcard (*) to indicate a summary of all such objects. The wildcard or name can be prefixed by an alias name, or for multischema databases a catalog and schema.

Names are by default in uppercase. If the object was defined in mixed or lower case, or with other special characters then use the SET DIALECT, or SET QUOTING RULES statements to enable delimited identifers. Then use quotes ("") around the name in the SHOW statement.

object-name
Specifies the name of an object whose definition you want to display.

ON DATABASE alias
Specifies the databases for which you want to display access privilege set information with the SHOW AUDIT, SHOW PRIVILEGES or SHOW PROTECTION statements. 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.

ON TABLES table-name
ON VIEWS view-name
ON COLUMNS column-name
ON FUNCTIONS function-name
ON PROCEDURES procedure-name
ON MODULES module-name
ON SEQUENCES sequence-name
Specifies the object for which you want to display access privilege set information with the SHOW AUDIT, SHOW PRIVILEGES or SHOW PROTECTION statements. You can specify a list of names, 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 table-name
Specifies the table or tables for which you want to see associated index definitions.

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, modules, or sequences.

  This statement displays not only the privileges that are explicitly granted to the user, but also any privileges that the user inherits from database access or the operating system.

  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, 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 of the entries in the access privilege set for the specified databases, tables, views, columns, external functions, external procedures, modules, or sequences.

PROCEDURES
Displays information about a specified procedure; either external or stored.

If you do not specify any of the SHOW PROCEDURES attributes (COMMENT, 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:
COMMENT	The description of the stored procedure. If none exists, nothing displays.
ID	The unique identification assigned to the procedure.
LANGUAGE	The language in which the procedure source is coded.
MODULE	The identification number of the module to which a procedure belongs.
OWNER	The owner of the procedure.

You Specify This:	SQL Displays Information About:
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.

PROFILES
Displays the definition of the specified profile. If you do not specify a wildcard or list of profile names, SQL displays the names of all the profiles in all attached databases.

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.

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.)

ROLES
Displays the definition of the specified role. SQL displays the role name, ID number, and any comments associated with the role definition.

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.

SEQUENCES
Displays the definition of the specified sequence. SQL displays the sequence name, ID number, and the sequence attributes.

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 C for more information about the SQLCA.

STATISTICS

Displays simple process statistics for the current process. This command is used primarily to compare resource usage and elapsed time for different queries.

The following example shows the output after performing a typical query:

```
SQL> select count (*)
cont> from employees natural full outer join job_history;

      274
1 row selected
SQL> show statistics;

                  process statistics at 5-MAR-2006 05:57:48.28
         elapsed time = 0 00:00:00.16              CPU time =  0 00:00:00.05
     page fault count = 430            pages in working set = 22768
   buffered I/O count = 26                 direct I/O count = 83
      open file count = 12             file quota remaining = 7988
           locks held = 138                 locks remaining = 16776821
      CPU utilization = 31.2%           AST quota remaining = 995
```

The statistics are reset after each execution of the SHOW STATISTICS command.

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.

You Specify This:	SQL Displays Information About:
USAGE	Usage, object name, storage map, and storage map partition number for the specified storage area. Partition numbers are always shown in parentheses, 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.
ATTRIBUTES	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.

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.

You Specify This:	SQL Displays Information About:
PARTITIONS	Storage map partitions showing the partition name, number and the name of the storage area used for the partition

SYNONYMS
Displays information about the specified synonyms. If you do not specify any aliases in the synonym names that you specify, SQL displays this information about all attached databases. The name of the target object, possibly another synonym, is displayed.

SYSTEM
ALL
Controls whether SQL displays system-defined domains, indexes, sequences, storage maps, tables, or views in the output of the SHOW statement.

- If you do not specify either SYSTEM or ALL, the display includes only user-defined elements.
- If you specify SYSTEM, the display includes elements created for use by the database system, or layered applications such as the OCI Services component of SQL/Services.
- If you specify ALL, the display includes both user-defined and system-defined elements.

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 (CARDINALITY, 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.

You Specify This:	SQL Displays Information About:
CARDINALITY	Displays the approximate cardinality as recorded in the Rdb system tables for the named tables and their indices.
COLUMNS	Each column name, data type, and domain name for the specified tables.
COMMENT	Comments for the specified tables.
CONSTRAINTS	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	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	Names of the storage maps for the specified tables.
TRIGGERS	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.

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.

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.

TRIGGERS
Displays information about the specified trigger. If you do not specify a wildcard or list of trigger names, SQL displays the names of all the triggers in all attached databases.

USERS
Displays the definition of the specified database user. SQL displays the database user name (such as defined by the CREATE USER statement), how the user will be authenticated (currently, only through the operating system), whether the account is locked or unlocked, and any comments associated with the user definition.

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.

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.

VARIABLES
Displays information about declared variables.

VERSIONS
Displays the version of SQL and the underlying software components.

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.

You Specify This:	SQL Displays Information About:
COLUMNS	Each column name, data type, and domain name for the specified views.
COMMENT	Comments for the specified views.
SOURCE	Source definitions for the specified views.

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.

Usage Notes

- The SET DISPLAY NO COMMENT statement will disable the display of COMMENT information by all SHOW commands.
- 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, parameter, or table is different than the database default character set, the SHOW 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.
- 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.
- If you use the ALTER TABLE statement to change the order in which columns are displayed, that ordering is also reflected when you issue a SHOW TABLE statement.
- If you issue a SHOW TABLES (CONSTRAINTS) statement, it indicates whether or not the constraint has been disabled.
- If you issue a SHOW TRIGGERS statement, it indicates whether or not the trigger has been disabled.
- The following usage notes apply to synonyms only:
  - If neither synonym name nor asterisk (*) is provided, then a list of all synonyms will be displayed with the type of object. If the word "synonym" appears in the description, then the source of this synonym is another synonym. In this case, use SHOW SYNONYM on the source object to get more information, otherwise use the appropriate SHOW statement for the named object.
  - If an asterisk (*) or a synonym name is specified then the synonym, its comment and details about the source object are displayed.
  - If a synonym is defined for a table, view, sequence, domain, module, procedure or function, then a SHOW for that type of object will also list the defined synonyms.
- The following SHOW commands allow the specified name to contain wildcard patterns that include "%", "_", and "\" (as the escape character) in order to select a subset of object names: SHOW COLLATING SEQUENCE, SHOW DOMAINS, SHOW FUNCTIONS, SHOW INDEXES, SHOW MODULES, SHOW OUTLINES, SHOW PROCEDURES, SHOW PROFILES,

SHOW ROLES, SHOW SEQUENCES, SHOW STORAGE MAPS, SHOW SYNONYMS, SHOW TABLES, SHOW TRIGGERS, SHOW USERS, and SHOW VIEWS.

For instance, the following query will display all tables with the string "JOB" in the name.

```
SQL> show table (comment) %JOB%
Information for table CURRENT_JOB

Comment on table CURRENT_JOB:
View to provide the current job for employees

Information for table JOBS

Comment on table JOBS:
Possible jobs in the company

Information for table JOB_HISTORY

Comment on table JOB_HISTORY:
Employment history within the company

SQL>
```

Note

This support is not currently available for multischema databases.

Refer to the documentation on the LIKE clause for information on the wildcard characters "%" and "_". For SHOW commands, the escape character is defined implicitly as "\".

- The following SHOW commands allow synonyms to be used to identify the object to be displayed: SHOW DOMAINS, SHOW FUNCTIONS, SHOW MODULES, SHOW PROCEDURES, SHOW SEQUENCES, SHOW TABLES, and SHOW VIEWS.

Note

This support is not currently available for multischema databases.

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
Display character set is UNSPECIFIED
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
Display character set is UNSPECIFIED

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
Display character set is UNSPECIFIED

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 ALIAS examples.
SQL> --
SQL> SHOW ALIAS;
Default alias:
    Oracle Rdb database in file MIA_CHAR_SET
Alias MIA1:
    Oracle Rdb database in file MIA_CHAR_SET
SQL> SHOW ALIAS MIA1;
Alias MIA1:
    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
        Number of nodes:              16
        Buffer Size (blocks/buffer):   6
        Number of Buffers:            20
        Number of Recovery Buffers:   20
        Snapshots are Enabled Immediate
   .
   .
   .
        ACL based protections
Storage Areas in database with alias MIA1
     RDB$SYSTEM                      Default and list storage area
Journals in database with alias MIA1
     No Journals Found
Cache Objects in database MIA1
     No Caches Found

SQL> --
SQL> -- SHOW CONNECTIONS examples.
SQL> --
SQL> CONNECT TO 'ALIAS MIA1 FILENAME MIA_CHAR_SET' AS 'TEST';
SQL> SHOW CONNECTIONS;
 RDB$DEFAULT_CONNECTION
-> TEST
SQL> SHOW CONNECTIONS DEFAULT;
Connection: RDB$DEFAULT_CONNECTION
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
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
Display character set is UNSPECIFIED

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 SMITH
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
Display character set is UNSPECIFIED

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 SQL99 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;
 RDB$DEFAULT_CONNECTION
 TEST
-> test1
SQL> SHOW CONNECTIONS test1;
Connection: TEST1
%SQL-F-NOSUCHCON, There is not an active connection by that name
SQL> SET QUOTING RULES 'SQL99';
SQL> SHOW CONNECTIONS "test1";
Connection: test1
Default alias is RDB$DBHANDLE
Default catalog name is RDB$CATALOG
Default schema name is SMITH
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
Display character set is UNSPECIFIED

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;
%SQL-I-SPELLCORR, identifier DATABASES replaced with DATABASE
Default alias:
    Oracle Rdb database in file MIA_CHAR_SET
Alias MIA1:
    Oracle Rdb database in file MIA_CHAR_SET
SQL> SHOW DATABASE RDB$DBHANDLE;
Default alias:
    Oracle Rdb database in file MIA_CHAR_SET
        Multischema mode is disabled
        Default character set is DEC_KANJI
        National character set is KANJI
        Identifier character set is DEC_KANJI
        Number of users:              50
        Number of nodes:              16
        Buffer Size (blocks/buffer):   6
        Number of Buffers:            20
        Number of Recovery Buffers:   20
        Snapshots are Enabled Immediate
   .
   .
   .
        ACL based protections
Storage Areas in database with filename MIA_CHAR_SET
     RDB$SYSTEM                      Default and list storage area
Journals in database with filename MIA_CHAR_SET
     No Journals Found
Cache Objects in database with filename MIA_CHAR_SET
     No Caches Found

SQL> --
SQL> -- SHOW DOMAINS example.
SQL> --
SQL> SHOW DOMAINS;
User domains in database with filename MIA_CHAR_SET
 No Domains Found
User domains in database with alias MIA1
 No Domains Found

SQL> --
SQL> -- SHOW TABLES example.
SQL> --
SQL> SHOW TABLES;
User tables in database with filename MIA_CHAR_SET
     COLOURS
User tables in database with alias MIA1
     MIA1.COLOURS
SQL> SHOW TABLE (COLUMNS) COLOURS;
Information for table COLOURS

Columns for table COLOURS:
Column Name   Data Type  Domain
-----------   ---------  ------
ENGLISH                         CHAR(8)
  DEC_MCS 8 Characters,  8 Octets
FRENCH                          CHAR(8)
  ISOLATIN9 8 Characters,  8 Octets
JAPANESE                        CHAR(8)
  SHIFT_JIS 4 Characters,  8 Octets
ROMAJI                          CHAR(16)
KATAKANA                        CHAR(8)
  KATAKANA 8 Characters,  8 Octets
HINDI                           CHAR(8)
  DEVANAGARI 8 Characters,  8 Octets
GREEK                           CHAR(8)
  ISOLATINGREEK 8 Characters,  8 Octets
ARABIC                          CHAR(8)
  ISOLATINARABIC 8 Characters,  8 Octets
RUSSIAN                         CHAR(8)
  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
User indexes in database with alias MIA1
     MIA1.COLOUR_INDEX
SQL> SHOW INDEXES COLOUR_INDEX;
Indexes on table COLOURS:
COLOUR_INDEX                    with column JAPANESE
  Duplicates are allowed
  Type is Sorted
  Key suffix compression is DISABLED
```

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
Output date and time format is:
  TIME =  15 (for example: kl 00.00)
Input date and time format is:
  tt-Monat-jjjj4 hh:mm:ss.cc2
SQL> --
SQL> SHOW DIGIT SEPARATOR
Digit separator is '.'.
SQL> --
SQL> SHOW LANGUAGE
Language is GERMAN.
SQL> --
SQL> SHOW RADIX POINT
Radix point is ','.
SQL>
SQL>disconnect all;
```

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
     AIJ_TWO
SQL> SHOW JOURNAL *
Journals in database with filename SAMPLE
     AIJ_ONE
        Journal File:    DISK1:[DOCS]AIJ1.AIJ;1
        Backup File:     DISK1:[DOCS.AIJS]AIJ1.AIJ;
     AIJ_TWO
        Journal File:    DISK1:[DOCS]AIJ2.AIJ;1
        Backup File:     DISK1:[DOCS.AIJS]AIJ2.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:      DISK1:[DOCS.WORK]JOBS.RDA;1
         Area Allocation:          402 pages
         Extent:         Enabled
         Area Extent Minimum:      99 pages
         Area Extent Maximum:      9999 pages
         Area Extent Percent:      20 percent
         Snapshot File:  DISK1:[DOCS.WORK]JOBS.SNP;1
         Snapshot Allocation:      100 pages
         Snapshot Extent Minimum:  99 pages
         Snapshot Extent Maximum:  9999 pages
         Snapshot Extent Percent:  20 percent
         Locking is Row Level
         No Cache Associated with Storage Area
         Thresholds are (70, 85, 95)
```

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
```

Example 10: Displaying a Sequence

```
SQL> SHOW SEQUENCE EMPIDS
     EMPIDS
 Sequence Id: 3
 Initial Value: 1
 Minimum Value: 1
 Maximum Value: 9223372036854775787
 Next Sequence Value: 1
 Increment by: 1
 Cache Size: 20
 Order
 No Cycle
 No Randomize
 Comment:       Sequence for employee IDs.
```

Example 11: Displaying a Role

```
SQL> SHOW ROLE SECRETARY
     SECRETARY
 Identified Externally
 Comment:       Role for the secretarial staff
```

Example 12: Displaying a User

```
SQL> SHOW USER NSTEWART
     NSTEWART
 Identified Externally
 Account Unlocked
 Comment:       Nicholas Stewart
```

Example 13: Show Details of One Profile

```
SQL> SHOW PROFILE
Profiles in database with filename SQL$DATABASE
     DECISION_SUPPORT
SQL> SHOW PROFILE DECISION_SUPPORT
     DECISION_SUPPORT
 Comment:       limit transactions used by report writers
     Transaction modes (read only, no read write)
SQL> ALTER PROFILE DECISION_SUPPORT
cont>   default transaction read only;
SQL> SHOW PROFILE DECISION_SUPPORT
     DECISION_SUPPORT
 Comment:       limit transactions used by report writers
     Default transaction read only
     Transaction modes (read only, no read write)
SQL>
```

Example 14: Show the Use of Delimited Identifiers for Mixed-Case Names

```
SQL> CREATE PROFILE "Decision_Support"
cont> COMMENT IS 'limit transactions used by report writers'
cont> TRANSACTION MODES (NO READ WRITE, READ ONLY);
SQL> SHOW PROFILE
Profiles in database with filename SQL$DATABASE
     Decision_Support
SQL> SHOW PROFILE Decision_Support
 No Users found
SQL> SHOW PROFILE "Decision_Support"
     Decision_Support
 Comment:       limit transactions used by report writers
     Transaction modes (read only, no read write)
```

Example 15: Displaying Synonyms

```
SQL> SHOW SYNONYMS
Synonyms in database with filename SQL$DATABASE
     C_SAL                           View                CURRENT_SALARY
     E                               Table synonym       EMPS
     EMPS                            Table               EMPLOYEES
     ID_NUMBER                       Domain              ID_DOM

SQL> SHOW SYNONYMS ID_NUMBER
 ID_NUMBER
 for domain  ID_DOM
 Comment:       support the old name for this domain

SQL> SHOW VIEWS

User tables in database with filename SQL$DATABASE
     CURRENT_INFO                    A view.
     CURRENT_JOB                     A view.
     CURRENT_SALARY                  A view.
     C_SAL                           A synonym for view CURRENT_SALARY
```

Example 16: Using Synonyms to Identify Objects

This example creates a sequence and a synonym for a sequence, and uses the SHOW SEQUENCE command with the synonym.

```
SQL> create sequence department_id_sequence;
SQL> create synonym dept_id_s for department_id_sequence;
SQL> show sequence
Sequences in database with filename personnel
     DEPARTMENT_ID_SEQUENCE
     DEPT_ID_S                       A synonym for sequence DEPARTMENT_ID_SEQUENCE
SQL> show sequence DEPT_ID_S
     DEPT_ID_S                       A synonym for sequence DEPARTMENT_ID_SEQUENCE
 Sequence Id: 1
 Initial Value: 1
 Minimum Value: 1
 Maximum Value: 9223372036854775787
 Next Sequence Value: 1
 Increment by: 1
Next Sequence Value: 1
 Increment by: 1
 Cache Size: 20
 No Order
 No Cycle
 No Randomize
 Wait
SQL>
```

Example 17: SHOW AUDIT command

The following example shows the output for SHOW AUDIT ON DATABASE. A list of two database ALIAS are specified, one database has auditing enabled, the other has no auditing.

```
SQL> SHOW AUDIT ON DATABASE RDB$DBHANDLE, DB1;
Audit information for Alias RDB$DBHANDLE
 Auditing is enabled
 Alarms will be written to the operator
 Audit every object access
 Forced writes of audit journal records is enabled
 Audit Event Classes:
         PROTECTION (Grant and Revoke)
         DACCESS (Discretionary Access)
 Identifiers:
        [DEV,TEST_EXECUTE]
        [PRD,*]
        [MGR,ADMIN]
        [AUD,*]
 Audit Privileges:
        SELECT,DISTRIBTRAN
 Alarm Privileges:
        DROP,SECURITY

Audit information for Alias DB1
 Auditing is disabled

SQL>
```

Example 18: SHOW TABLE (CARDINALITY) command

The following example shows the additional output when the CARDINALITY option is used with SHOW TABLES.

```
SQL> show table (cardinality,index) salary_history;
Information for table SALARY_HISTORY

  Table cardinality: 729
```

```
Indexes on table SALARY_HISTORY:
SH_EMPLOYEE_ID                  with column EMPLOYEE_ID
  Index cardinality: 100
  Duplicates are allowed
  Type is Sorted
  Key suffix compression is DISABLED
  Node size 430
  Percent fill 70
```

SIGNAL Control Statement

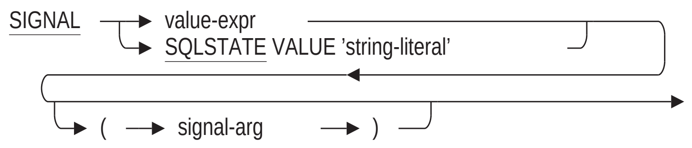
Passes the signaled SQLSTATE status parameter back to the application or SQL interface and terminates the current routine and all calling routines.

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



Arguments

signal-arg
Specifies a value expression. The specified value is converted to a CHARACTER(80) CHARACTER SET UNSPECIFIED string and returned as a secondary message to the client application. If the value expression converts to a character string longer than 80 characters, it is truncated.

You can use the sql_get_error_text routine to extract the signal-arg text in an application.

string-literal
A quoted string literal which represents the SQLSTATE value.

value-expr
Expects a 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 and triggers 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, DEC_MCS, ISOLATIN1, or ISOLATIN9.
- A numeric value expression used with SIGNAL 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
```

- The error message returned by Oracle Rdb includes the name of the routine or trigger that called SIGNAL. 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"
```

Note

You can provide a name for a compound statement using the OPTIMIZE AS clause in the BEGIN or PRAGMA clause.

- 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

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").

```
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"
```

Example 2: Specifying a Secondary Error

```
SQL> BEGIN
SQL> SIGNAL SQLSTATE 'RR000' (' Compound Statement Failed');
cont> END;
%RDB-E-SIGNAL_SQLSTATE, routine "(unnamed)" signaled SQLSTATE "RR000"
-RDB-I-TEXT,  Compound Statement Failed
```

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 1.4 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;
```

START TRANSACTION Statement

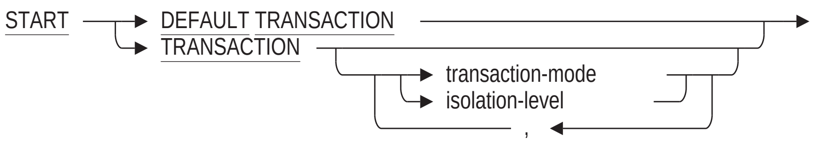
Starts a transaction using the specified attributes. If DEFAULT is specified, then the attributes are derived from the user's profile.

Environment

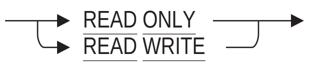
You can use the START TRANSACTION statement:

- 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

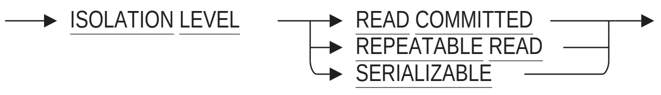
Format



transaction-mode =



isolation-level =



Arguments

DEFAULT
If the keyword DEFAULT is used, the user-specific default transaction is started. This default is defined in the profile for the current session user. If none is specified, a READ ONLY transaction will be started.

```
SQL> CREATE PROFILE READ_USERS
cont>   DEFAULT TRANSACTION READ ONLY WAIT 10;
SQL> ALTER USER JONES PROFILE READ_USERS;
```

A START DEFAULT TRANSACTION statement executed by JONES will start a READ ONLY WAIT 10 transaction.

For information on profiles see the ALTER and CREATE PROFILE statements.

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 8–11 shows the phenomena permitted for the isolation levels that you can explicitly specify with the START TRANSACTION statement.

Table 8–11 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 Rdb Guide to SQL Programming* for further information about specifying isolation levels in transactions.

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 are implicitly isolation level serializable.

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 to maintain isolation level serializable.
- If your query uses indexes, SQL may lock part of an index, which has the effect of locking several rows.

Usage Notes

- The START TRANSACTION statement is similar to the SET TRANSACTION statement in operation. That is, you can specify READ WRITE or READ ONLY transaction modes as well as various isolation levels.
- The transaction-mode and isolation-level clauses may appear only once in any START TRANSACTION statement.
- This statement does not support BATCH UPDATE mode, as this is an Oracle Rdb extension and, therefore, is only supported by SET and DECLARE TRANSACTION statements.
- Oracle Rdb has extended the START TRANSACTION statement and allows all transaction options to be omitted. If the transaction-mode is omitted, it defaults to READ WRITE. If the isolation-level is omitted, it defaults to ISOLATION LEVEL SERIALIZABLE. Therefore, if all options are omitted, the transaction defaults to READ WRITE ISOLATION LEVEL SERIALIZABLE.
- If more than one database is currently attached, a transaction spanning all databases will be started with the specified or default attributes.
- You cannot use the START TRANSACTION statement in an ATOMIC compound statement.

- The START TRANSACTION statement may not be executed from a SQL function or trigger or any stored procedure called from a SQL function or trigger.

Examples

Example 1: Starting a Default Transaction in a Multistatement Procedure or as a Single Statement

```
SQL> START DEFAULT TRANSACTION;
SQL>
SQL> BEGIN
cont> COMMIT;
cont> START DEFAULT TRANSACTION;
cont> END;
SQL>
SQL> ROLLBACK;
```

Example 2: Starting Several Variations of the START TRANSACTION Statement

```
SQL> START TRANSACTION READ WRITE,
cont>   ISOLATION LEVEL READ COMMITTED;
SQL> COMMIT;
SQL>
SQL> -- Defaults to serializable
SQL> START TRANSACTION READ WRITE;
SQL> COMMIT;
SQL>
SQL> -- Defaults to read write
SQL> START TRANSACTION ISOLATION LEVEL READ COMMITTED;
SQL> ROLLBACK;
SQL>
SQL> -- Defaults to read write serializable
SQL> START TRANSACTION;
SQL>
SQL> BEGIN
cont> COMMIT;
cont> START TRANSACTION
cont>     ISOLATION LEVEL READ COMMITTED,
cont>     READ WRITE;
cont> END;
SQL> COMMIT;
```

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.

Trace logging can help you debug complex multistatement procedures.

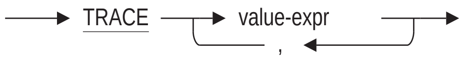
Environment

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.
- The TRACE statement is enabled by one of these methods:
  - SET FLAGS 'TRACE' statement
  - Defining the RDMS$SET_FLAGS logical name including the 'TRACE' keyword
  - Defining the RDMS$DEBUG_FLAGS logical name including the Xt string (note that X is uppercase and t is lowercase)

  Output can be redirected using the RDMS$DEBUG_FLAGS_OUTPUT logical name. See Appendix E and the *Oracle Rdb7 Guide to Database Performance and Tuning* for information on logical names.

- 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
```

- If the TRACE statement is activated then queries in the TRACE statement are merged into the query outline for the procedure. Example 2 in the Examples section shows a query outline that contains one query when the TRACE statement is disabled.
- If any TRACE statement contains a subquery, then Oracle Corporation recommends using two query outlines (if any are required at all), with different modes in order to run the query with and without TRACE enabled. That is, when TRACE is enabled, define MODE to match the TRACE enabled query outlines.

```
$ DEFINE RDMS$DEBUG_FLAGS_OUTPUT TRACE.DAT
$ DEFINE RDMS$SET_FLAGS "TRACE, MODE(10)"
```

  Alternatively, use the SET FLAGS statement, which allows the TRACE flag to be enabled and the MODE established from within an interactive session or through dynamic SQL. This method allows the query to be run with TRACE enabled or disabled.
- Use the COALESCE function to format NULL expressions. For example, TRACE COALESCE(LAST_NAME, 'NULL');.

Examples

Example 1: Tracing a multistatement procedure

```
SQL> ATTACH 'FILENAME MF_PERSONNEL';
SQL> SET FLAGS 'TRACE';
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
```

Example 2: Generating a query outline when the TRACE statement is disabled

```
SQL> DECLARE :LN CHAR(40);
SQL> SET FLAGS 'NOTRACE';
SQL> BEGIN
cont> TRACE 'Jobs Held: ',
cont>     (SELECT COUNT(*)
cont>         FROM JOB_HISTORY
cont>         WHERE EMPLOYEE_ID = '00201');
cont> SELECT LAST_NAME
cont>     INTO :LN
cont>     FROM EMPLOYEES
cont>     WHERE EMPLOYEE_ID = '00201';
cont> END;
-- Oracle Rdb Generated Outline :
create outline QO_A17FA4B41EF1A68B_00000000
id 'A17FA4B41EF1A68B966C1A0B083BFDD4'
mode 0
as (
  query (
-- Select
    subquery (
      EMPLOYEES 0     access path index       EMPLOYEES_HASH
      )
    )
  )
compliance optional     ;
SQL>
```

If the query outline is generated with TRACE enabled, then two queries appear; the first is for the subquery in the TRACE statement and the other is for the singleton SELECT statement.

If this second query outline is used at run time with the TRACE statement disabled, then it cannot be applied to the query. Because the outline was created with compliance optional, the query outline is abandoned and a new strategy is calculated. If compliance is mandatory, then the query fails. See Example 3.

```
SQL> DECLARE :LN CHAR(40);
SQL> SET FLAGS 'TRACE';
SQL> BEGIN
cont> TRACE 'Jobs Held: ',
cont>     (SELECT COUNT(*)
cont>         FROM JOB_HISTORY
cont>         WHERE EMPLOYEE_ID = '00201');
cont> SELECT LAST_NAME
cont>     INTO :LN
cont>     FROM EMPLOYEES
cont>     WHERE EMPLOYEE_ID = '00201';
cont> END;
-- Oracle Rdb Generated Outline :
create outline QO_A17FA4B41EF1A68B_00000000
id 'A17FA4B41EF1A68B966C1A0B083BFDD4'
mode 0
as (
  query (
-- Trace
    subquery (
      JOB_HISTORY 0   access path index       JOB_HISTORY_HASH
      )
    )
  query (
-- Select
    subquery (
      EMPLOYEES 0     access path index       EMPLOYEES_HASH
      )
    )
  )
compliance optional     ;
~Xt: Jobs Held: 4
SQL>
```

Example 3: Using an Outline with Tracing Enabled That Was Created with Tracing Disabled

This example shows that enabling the TRACE statement may affect query outlines defined when TRACE was disabled.

```
SQL> DECLARE :LN CHAR(40);
SQL>
SQL> BEGIN
cont> TRACE 'Jobs Held: ',
cont>     (SELECT COUNT(*)
cont>         FROM JOB_HISTORY
cont>         WHERE EMPLOYEE_ID = '00201');
cont> SELECT LAST_NAME
cont>     INTO :LN
cont>     FROM EMPLOYEES
cont>     WHERE EMPLOYEE_ID = '00201';
cont> END;
~S: Outline QO_A17FA4B41EF1A68B_00000000 used
~S: Outline/query mismatch; assuming JOB_HISTORY 0 renamed to EMPLOYEES 0
~S: Full compliance with the outline was not possible
Get     Retrieval by index of relation EMPLOYEES
  Index name  EMPLOYEES_HASH [1:1]       Direct lookup
```

TRUNCATE TABLE Statement

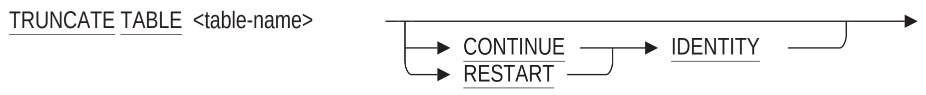
Deletes the data in a table while still maintaining the metadata definitions of the table. Advantages include fast deletion of data in uniform areas, and no change to dependency data.

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



Arguments

table-name
Specifies the name of the table you want to truncate. This name must be a base table, or global temporary table. Views and local temporary tables may not be truncated.

CONTINUE IDENTITY
Requests that the TRUNCATE TABLE statement leave the current next value unchanged for the associated IDENTITY column.

RESTART IDENTITY
Requests that TRUNCATE TABLE reset the associated IDENTITY column so that it starts with the START WITH value, or, if there is none, the MINVALUE value defined for the sequence.

Usage Notes

- You must have DELETE privilege for the table as this commands deletes all data.
- You must have CREATE privilege at the table level.
- If there exists an AFTER DELETE or BEFORE DELETE trigger defined on this table, you will require DROP and CREATE privileges for triggers on this table. These privileges are required because this operation effectively disables these triggers.
- TRUNCATE TABLE is a data definition statement and as such requires exclusive access to the table.

- The TRUNCATE TABLE statement fails with an error message if:
  - RDB$SYSTEM storage area is set to read-only
  - The named table is a view
  - The named table has been reserved for data definition
  - The named table is a system table
- TRUNCATE TABLE deletes all data in the table, however, it does not execute any BEFORE or AFTER DELETE triggers.
- If the dialect is set to SQL2011 and neither CONTINUE IDENTITY nor RESTART IDENTITY clauses are specified, the default will be CONTINUE IDENTITY. For all other dialects, the default is RESTART IDENTITY.
- TRUNCATE TABLE explicitly resets the values in Rdb$WORKLOAD rows associated with this table, as well as removing any index or table storage statistics.
- All CHECK and FOREIGN KEY constraints that reference the truncated table are revalidated 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> set dialect 'sql99';
  SQL> CREATE TABLE test1
  cont>  (col1 REAL PRIMARY KEY);
  SQL> CREATE TABLE test2
  cont>  (col1 REAL REFERENCES TEST1 (COL1));
  SQL> INSERT INTO test1 VALUES (1);
  1 row inserted
  SQL> INSERT INTO test2 VALUES (1);
  1 row inserted
  SQL> COMMIT;
  SQL>
  SQL> TRUNCATE TABLE test1;
  %RDB-E-NO_META_UPDATE, metadata update failed
  -RDB-E-INTEG_FAIL, violation of constraint TEST2_FOREIGN1 caused operation to
  fail
  -RDB-F-ON_DB, on database USERS2:[TESTING.DATABASES.]PERSONNEL.RDB;1
  SQL> TABLE test1;
             COL1
    1.0000000E+00
  1 row selected
  SQL> ROLLBACK;
  ```
- When a table contains one or more LIST OF BYTE VARYING columns, the TRUNCATE TABLE statement must read each row in the table and record the pointers for all LIST values. This list is processed at COMMIT time to delete the LIST column data. Therefore, the database administrator must also allow for this time when truncating the table.

  Reserving the table for EXCLUSIVE WRITE is recommended because the dropped LIST columns will require that each row in the table be updated and set to NULL - it is this action which queues the pointers for commit time processing. This reserving mode will eliminate snapshot file I/O, lower lock resources and reduce virtual memory usage.

As the LIST data is stored outside the table, performance may be improved by attaching to the database with the RESTRICTED ACCESS clause, which has the side effect of reserving all the LIST storage areas for EXCLUSIVE access and therefore eliminates snapshot I/O during the delete of the LIST data.

- If the table contains no LIST OF BYTE VARYING columns, and the table and all associated indices are stored in UNIFORM storage areas, then TRUNCATE TABLE will employ the most efficient mechanism to erase the data from the table.

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
```

Example 2: Using the Continue Identity clause

The following example requests that the IDENTITY column not be restarted after the truncate.

```
SQL> truncate table HISTORY_LOG continue identity;
```

UNDECLARE Cursor Statement

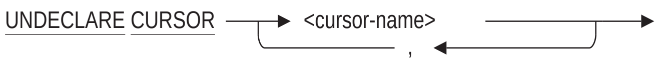
This statement implicitly closes the named cursor, removes the declared cursor name from the known cursor list, and releases resources held by SQL and the Oracle Rdb Server for that cursor.

Environment

You can use the UNDECLARE CURSOR statement:

- In interactive SQL

Format



Arguments

cursor-name
Specifies the name of the declared cursor.

Usage Notes

- If this is a table cursor, then all associated list cursors are also undeclared.

Example

Example 1: Using the Undeclare Cursor statement

This example demonstates the use of the UNDECLARE CURSOR Statement.

```
SQL> declare mycursor cursor for select * from work_status;
SQL>
SQL> open mycursor;
SQL> fetch mycursor;
 STATUS_CODE   STATUS_NAME   STATUS_TYPE
 0             INACTIVE      RECORD EXPIRED
SQL> close mycursor;
SQL>
SQL> undeclare cursor mycursor;
SQL>
SQL> --> expect an error
SQL> open mycursor;
%SQL-F-NOSUCHCUR, Cursor MYCURSOR has not been declared
SQL>
```

UNDECLARE Variable Statement

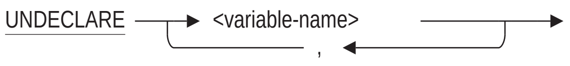
Deletes a variable definition from interactive and dynamic SQL that was used for invoking stored procedures and for testing procedures in modules or embedded SQL programs.

Environment

You can use the UNDECLARE statement:

- In interactive SQL
- In dynamic SQL as a statement to be dynamically executed

Format



Arguments

variable-name
Specifies the name of the local variable prefixed with a colon (:).

Usage Notes

- Local variables will remain until a successful UNDECLARE statement, or until the image runs down.

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

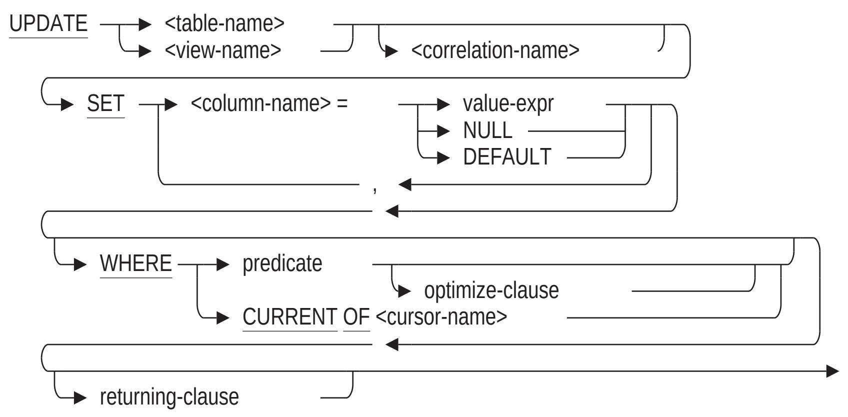
Modifies a row in a table or view.

Environment

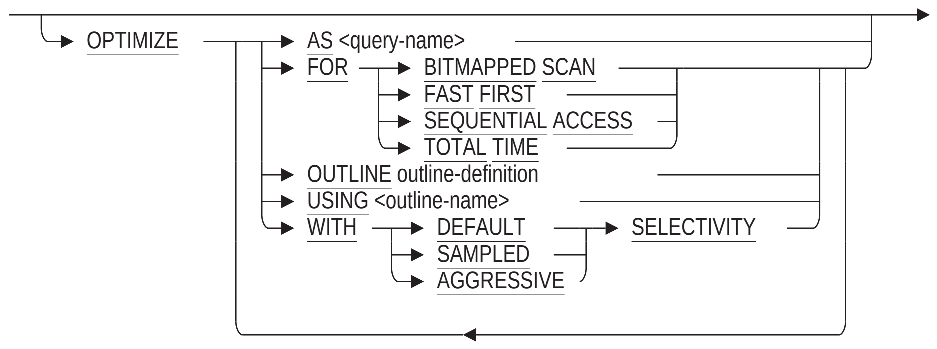
You can use the UPDATE statement:

- 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

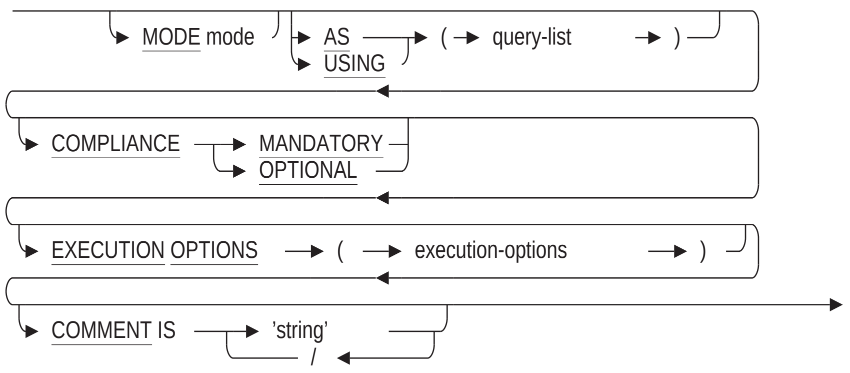
Format



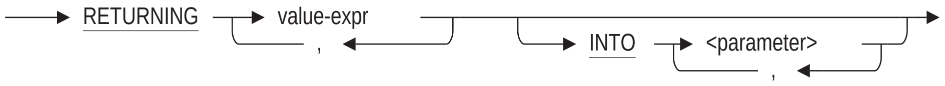
optimize-clause =



outline-definition =



returning-clause=



Arguments

column-name
Specifies the name of a column whose value 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 2.2.4.1 for more information about correlation names.

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.

DEFAULT
SQL assigns the DEFAULT defined for the column or domain. If no DEFAULT is defined, then NULL is assumed.

If the DEFAULT clause is used in an UPDATE statement then one of the following will be applied:

- If a DEFAULT attribute is present for the column then that value will be applied during UPDATE.

- Else if an AUTOMATIC attribute is present for the column then that value will be applied during UPDATE. This can only happen if the SET FLAGS 'AUTO_OVERRIDE' is used since during normal processing these columns are read-only.
- Otherwise a NULL will be applied during UPDATE.

INTO parameter
Inserts the values specified to specified parameters.

The INTO parameter clause is optional in interactive SQL. In this case the returned values are displayed.

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).

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

OPTIMIZE FOR
The OPTIMIZE FOR clause specifies the preferred optimizer strategy for statements that specify a select expression. The following options are available:

- BITMAPPED SCAN

  Requests the Rdb query optimizer attempt to use BITMAPPED SCAN if there exists multiple supporting indices. This option is not compatible with SEQUENTIAL ACCESS.
- 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.
- SEQUENTIAL ACCESS

  Forces the use of sequential access. This is particularly valuable for tables that use the strict partitioning functionality.
- 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 OUTLINE outline-definition
The OPTIMIZE OUTLINE clause declares a temporary query outline to be used with the select expression.

See the CREATE OUTLINE Statement for more information on defining an outline.

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 WITH
Selects one of three optimzation controls: DEFAULT (as used by previous versions of Rdb), AGGRESSIVE (assumes smaller numbers of rows will be selected), and SAMPLED (which uses literals in the query to perform preliminary estimation on indices).

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.

RETURNING value-expr
Returns the listed value expressions. 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.

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, the NULL keyword, or the DEFAULT keyword. SQL assigns the value following the equal sign to the column that precedes the equal sign.

table-name
view-name
Specifies the name of the target table or view that 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 values can be specified through parameters, qualified parameters, column select expressions, value expressions, or the default values.

See Chapter 2 for more information about parameters, qualified parameters, column select expressions, value expressions, and default values.

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.

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 both the OPTIMIZE clause and 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
```

Example 3: Using UPDATE ... WHERE CURRENT OF with a compound statement FOR loop

This compound statement uses a FOR loop to update each employee to apply a company wide pay raise. The FOR loop defines a cursor name (SAL_CURSOR) that fetches each row of the employees current salary. The nested UPDATE statement uses "where current of SAL_CURSOR" to apply the update to the current row of the cursor.

```
SQL> begin
cont> for :sal
cont>     as each row of table cursor SAL_CURSOR
cont>     for select salary_amount
cont>         from salary_history
cont>         where salary_end is null
cont> do
cont>     update salary_history
cont>     set salary_amount = salary_amount * 1.001
cont>     where current of SAL_CURSOR;
cont> end for;
cont> commit;
cont> end;
```

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:

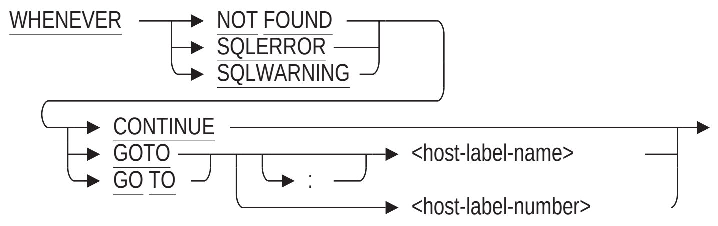
- Row not found
- An error condition
- A warning condition

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

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.

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 C for a list of the conditions that result in negative values for the SQLCODE field.

SQLWARNING
Indicates any warning condition. Appendix C lists the conditions that result in warnings for the SQLSTATE Status Parameter.

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 current ANSI/ISO SQL standard does not allow this colon.

Example

Example 1: Using WHENEVER statements in a C or C++ program

```
/* When an SQL statement results in
an end of stream status, 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;
```

```
/* Declare the cursor */
   .
   .
   .
/* Open the cursor */
exec sql
    OPEN MANAGER;

/* Start a loop to process the rows of the cursor */
for (;;)
{
    /* 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 */
    printf ("%s %s %s\n", fname, lname, dname);
    recs++;
}

label_not_found:
    printf ("%d rows\n", recs);
   .
   .
   .
exit (EXIT_SUCCESS);

label_error:
    sql_signal ();
    exit (EXIT_FAILURE);
   .
   .
   .
```

WHILE Control Statement

Allows the repetitive execution of one or more SQL statements in a compound statement based on the truth value of a predicate.

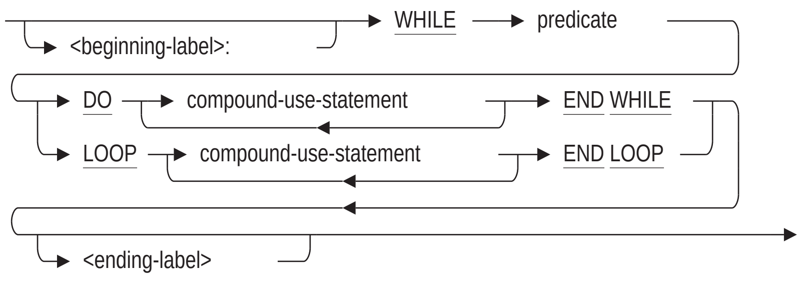
Environment

You can use the WHILE 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

while-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 the WHILE loop. If you include an ending label, it must be identical to its corresponding beginning label. A beginning label must be unique within the procedure containing the label.

compound-use-statement
Identifies the SQL statements allowed in a compound statement block. See the Compound Statement for the list of valid statements.

DO
Marks the start of a control loop.

END LOOP ending-label
Marks the end of a LOOP 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 ending-label argument makes multistatement procedures easier to read, especially in very complex multistatement procedure blocks.

END WHILE ending-label
Marks the end of a DO 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 ending-label argument makes multistatement procedures easier to read, especially in very complex multistatement procedure blocks.

LOOP
Marks the start of a control loop.

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. 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-DO statement executes the SQL statements embedded within its DO . . . END WHILE block. If the search condition evaluates to FALSE or UNKNOWN, SQL bypasses the DO . . . END WHILE block and passes control to the next statement.

Usage Notes

Although the DO . . . END WHILE and LOOP . . . END LOOP are semantically equivalent, the DO . . . END WHILE syntax conforms to the ANSI/ISO SQL/PSM standard.

Examples

Example 1: Using the While Statement to Count Substrings

```
SQL> DECLARE :SUB_STR CHAR;
SQL> DECLARE :SRC_STR CHAR(50);
SQL> BEGIN
cont>   SET :SUB_STR='l';
cont>   SET :SRC_STR='The rain in Spain falls mainly on the plain';
cont> END;
SQL> SET FLAGS 'TRACE';
SQL> BEGIN
cont>-- This procedure counts the occurrence of substrings
cont>   DECLARE :STR_COUNT INTEGER=0;
cont>   DECLARE :CUR_POS INTEGER = POSITION (:SUB_STR IN :SRC_STR);
cont>   WHILE :CUR_POS >0 DO
cont>      SET :STR_COUNT=:STR_COUNT + 1;
cont>         SET :CUR_POS = POSITION (:SUB_STR IN :SRC_STR FROM :CUR_POS + 1);
cont>   END WHILE;
cont>     TRACE 'FOUND ', :STR_COUNT, ' OCCURRENCES OF "', :SUB_STR, '"';
cont> END;
~Xt: Found 4          occurrences of "l"
```

Index