Oracle Rdb™

SQL Reference Manual Volume 1

Oracle Rdb Release 7.1.4.3 for OpenVMS Alpha

December 2005

ORACLE®

SQL Reference Manual, Volume 1

Oracle Rdb Release 7.1.4.3 for OpenVMS Alpha

Contents

3 SQL Module Language

4 SQL Precompiler

5 SQL Routines

Index

Examples

Figures

Tables

Send Us Your Comments

Oracle Rdb for OpenVMS
Oracle SQL Reference Manual, Release 7.1.4.1

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, chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail:nedc-doc_us@oracle.com
- FAX — 603-897-3825 Attn: Oracle Rdb
- Postal service:
  Oracle Corporation
  Oracle Rdb Documentation
  One Oracle Drive
  Nashua, NH 03062-2804
  USA

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

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

Preface

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

Intended Audience

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

Operating System Information

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

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 contains Chapter 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 ANSI standard, how to read syntax diagrams, executable and nonexecutable statements, keywords and line terminators, and support for Multivendor Integration Architecture (MIA).
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 convergence with Oracle7 SQL. This appendix also describes the SQL syntax for performing an outer join between tables.
Appendix H	Describes information tables that can be used with Oracle Rdb.
Index	Index for each volume.

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 7.1 of Oracle Rdb software is often referred to as V7.1.
- Oracle CDD/Repository software is referred to as the dictionary, the data dictionary, or the repository.
- Oracle ODBC Driver for Rdb software is referred to as the ODBC driver.
- OpenVMS means the OpenVMS Alpha operating system.

Technical Changes and New Features

Oracle Rdb Oracle Rdb Release 7.1.4.3 for OpenVMS Alpha has been enhanced in many areas for this release. The following highlights these new features.

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

- General changes
  - New spelling corrector for interactive SQL
  - The output of many SHOW commands has changed to make the output easier to read and to accommodate new features such as synonyms
  - The COMMENT ON statement has been enhanced to support all database objects, including the database itself.
- New objects that can be created
  - PROFILES
  - ROLES
  - SEQUENCES
  - SYNONYMS
  - USERS

  Each object type is supported by a CREATE, ALTER, DROP, RENAME and SHOW statement
- New ALTER statements for existing objects
  - ALTER CONSTRAINT
  - ALTER FUNCTION
  - ALTER MODULE
  - ALTER OUTLINE
  - ALTER PROCEDURE
  - ALTER TRIGGER

- New built in functions
  - ABS
  - LEAST
  - GREATEST
  - LOCALTIME
  - LOCALTIMESTAMP
  - UID
  - CURRENT_UID
  - SESSION_UID
  - SYSTEM_UID
  - TRANSLATE allows character string processing
  - SIZEOF (and VSIZE)
  - LENGTH is a synonym for CHAR_LENGTH
  - LENGTHB is a synonym for OCTET_LENGTH
  - BITSTRING
- New statistical functions
  - STDDEV
  - VARIANCE
  - STDDEV_POP
  - STDDEV_SAMP
  - VAR_POP
  - VAR_SAMP

  Also support for the FILTER qualifier on statistical functions
- New keywords for the GET DIAGNOSTICS statement
  - HOT_STANDBY_MODE
  - IMAGE_NAME
  - SERVER_IDENTIFICATION
  - TRANSACTION_CHANGE_ALLOWED
  - TRANSACTION_SEQUENCE

- TRANSACTION_TIMESTAMP

- New keywords for the SET FLAGS statement
  - ALTERNATE_OUTLINE_ID
  - AUTO_INDEX
  - AUTO_OVERRIDE
  - BITMAPPED_SCAN
  - CARTESIAN_LIMIT
  - COSTING
  - COUNT_SCAN
  - CURSOR_STAT
  - DETAIL_LEVEL
  - ESTIMATES
  - INDEX_DEFER_ROOT
  - LAREA_READY
  - MAX_RECURSION
  - MAX_SOLUTION
  - SEQ_CACHE
  - SELECTIVITY
  - SOLUTIONS
  - TEST_SYSTEM
  - TRANSITIVITY
  - VARIANCE_DOF
  - WARN_DDL
  - WARN_INVALID
  - WATCH_CALL
  - WATCH_OPEN
- New operators
  - EXCEPT DISTINCT operator
  - MINUS operator

- INTERSECT DISTINCT operator
- New UNIQUE predicate

- Support for Oracle style outer join syntax
- New optional system tables

  Oracle Rdb now supports Information tables which are read-only tables that represent Rdb internal data in tabular format:

  - RDB$CACHES
  - RDB$CHARACTER_SETS
  - RDB$DATABASE_JOURNAL
  - RDB$DATABASE_ROOT
  - RDB$DATABASE_USERS
  - RDB$JOURNALS
  - RDB$LOGICAL_AREAS
  - RDB$NLS_CHARACTER_SETS
  - RDB$STORAGE_AREAS

  When ALTER DATABASE . . . SYNONYMS ARE ENABLED is executed a new RDB$OBJECT_SYNONYMS table is created

  Users, roles and profiles use two new tables:

  - RDB$PROFILES
  - RDB$GRANTED_PROFILES

- New and changed statements
  - ACCEPT
  - DECLARE FUNCTION
  - DECLARE PROCEDURE
  - The DECLARE variable statement in interactive SQL now supports DEFAULT and CONSTANT clauses
  - GET ENVIRONMENT statement to translate symbols, logical name and to return session information
  - RENAME command and related RENAME TO clauses for many ALTER statements

- SET DISPLAY statement controls the output of SELECT and other statements
- SET PAGE LENGTH
- SET SESSION AUTHORIZATION
- SET CONTINUE CHARACTER
- SET DEFAULT CONSTRAINT MODE
- START TRANSACTION
- START DEFAULT TRANSACTION
- DECLARE DEFAULT TRANSACTION
- COMMIT AND CHAIN
- ROLLBACK AND CHAIN
- LOCK TABLE statement
- ITERATE statement
- REPEAT statement
- FOR counted loop
- Searched CASE statement
- Simple CASE statement allows list of values in the WHEN clause
- SIGNAL statement now allows an optional text string argument which is returned to the calling application
- PRAGMA clause for compound statements support WITH HOLD, etc
- new NAME clause for SET and DECLARE TRANSACTION statements
- INSERT now supports the DEFAULT VALUES clause
- INSERT . . . FROM FILENAME allows interactive SQL to load LIST OF BYTE VARYING columns directly from a text file.
- SELECT statement allows EDIT USING clause for format control
- TABLE clause is now a shorthand for SELECT * FROM
- New GRANT and REVOKE format for user and role management
- SET ECHO, SET HEADING, SET FEEDBACK and SET NULL added for Oracle SQL*Plus compatibility

• Partition naming and management.

All index and storage map partitions can be named. This allows many per partition statements which allow easier index management.

- ADD PARTITION (hashed index only)
- DROP PARTITION (hashed index only)
- RENAME PARTITION
- MOVE PARTITION (hashed index only)
- TRUNCATE PARTITION
- BUILD PARTITION
- REBUILD PARTITION

• Index and storage map attributes
  - NOLOGGING
  - PARTITION clause used to name the partition
  - DUPLICATES ARE ALLOWED
  - MAINTENANCE IS ENABLED
  - PREFIX CARDINALITY COLLECTION
• Changes for tables
  - Ability to enable and disable constraints and triggers
  - Reorder column display (BEFORE and AFTER COLUMN clauses)
  - Implied data type through REFERENCES foreign key constraint
  - AUTOMATIC column definition
  - IDENTITY attributes
  - DEFAULT can be any value expression
  - Table and column constraints can be deleted with DROP CONSTRAINT statement
  - New INITIALLY IMMEDIATE and INITIALLY DEFERRED clauses added for constraints
  - The new LIKE clause allows CREATE TABLE to copy metadata from another table
• Changes for triggers
  - Support for TRACE, SIGNAL and CALL statements

- ALTER TRIGGER . . . COMPILE allows triggers to be compiled if metadata has been changed
- Triggers can be enabled or disabled using ALTER TABLE or ALTER TRIGGER

- Changes for Modules and Routines
  - Modules can now declare global variables which are shared by all routines in the module
  - Modules can now contain both external and stored routines
  - Routine parameters can include DEFAULT value and COMMENT descriptions
  - The LEAVE statement now allows the label to be omitted
  - The LANGUAGE SQL clause is now optional
  - The DETERMINISTIC and NOT DETERMINISTIC clauses are added as a synonyms for the NOT VARIANT and VARIANT clauses
  - External routines can be defined with CALLED ON NULL INPUT and NULL USED ON NULL INPUT clauses. These options control how an external function is invoked when one or more of the arguments is NULL.
  - A routine can be declared LOCAL so that they may only be referenced within the declaring module
  - Alternate WHILE syntax to conform to the SQL standard
  - Compound statements can now include the PRAGMA clauses such as OPTIMIZE USING, WITH HOLD, etc. to allow better tuning of stored routines.
- New Dialects
  - SQL99 - CORE level SQL:1999 standard
  - ORACLE LEVEL2 - includes SQL99 and Oracle semantics for NULL and zero length strings

  These are supported for SET DIALECT, SET QUOTING RULES, SET DEFAULT DATE FORMAT, etc
- New data types added for compatibility with Oracle RDBMS and SQL:1999
  - NUMBER
  - LONG

  - LONG RAW
  - RAW
  - VARCHAR2 and CHARACTER VARYING are now synonyms for VARCHAR
- New character sets and related support
  - SET AUTOMATIC TRANSLATION
  - SET DISPLAY CHARACTER SET
  - Arabic, Unicode, UTF8
- New database attributes for use with ALTER, CREATE and IMPORT DATABASE
  - LOGMINER SUPPORT
  - GALAXY SUPPORT
  - LARGE MEMORY support for Global Buffers
  - NOTIFY is now a database attribute
  - SINGLE INSTANCE option for NUMBER OF CLUSTER NODES clause
  - SHARED MEMORY IS PROCESS RESIDENT clause

1
Introduction to SQL Syntax

SQL (structured query language) is a data definition and data manipulation language for relational databases. Using SQL, you can create the data definitions (the **schema**) that comprise the **database**, store data in a database, and update both data and data definitions.

Most major vendors offer variations of SQL for their relational database products. This fact often makes SQL the preferred interface at sites using relational database products from a variety of vendors. In the Oracle Rdb documentation set, SQL refers to the Oracle Rdb implementation of the SQL standard ANSI/ISO/IEC 9075-1:1999, commonly referred to as the ANSI/ISO SQL standard. Oracle Rdb mostly conforms to the CORE of the ANSI/ISO SQL standard. Please refer to the Rdb Release Notes for the small list of SQL:1999 CORE features not currently supported by Oracle Rdb.

Relational database organization represents data in a two-dimensional format that SQL calls **tables**. Tables in a relational database are similar to printed tables. In SQL terminology, tables consist of a set of **rows** and **columns**. The columns, which usually have names, divide each row into a set of individual pieces of data.

1.1 Using SQL

You can use SQL statements in two ways:

- Interactively

  You can use interactive SQL to learn how the SQL statements work, test your data manipulation statements, and perform queries.

  Invoke interactive SQL using the following commands:

  ```
  $ ! Define a symbol for SQL:
  $ SQL$ == $SQL$
  $ ! Invoke SQL:
  $ SQL$
  ```

In prior versions of Rdb the keyword abbreviation and matching support in interactive SQL would discard extraneous characters from a token if an expected keyword matched the leading prefix. This was confusing in some cases and interactive SQL now generates an informational message to clearly state the substitution.

The following example shows the information message generated when extra characters are trimmed from the keyword.

```
SQL> create trigger mytrigger
cont>     after updatete on mytable2
%SQL-I-SPELLCORR, identifier UPDATETE replaced with UPDATE
cont>     (insert into mytable values (mytable2.a, 'Any', 'Value'))
cont>     for each row;
```

- In high-level language programs

  In an application program, you can use the same statements that you used interactively. There are two ways to do this:

  – You can embed statements directly in the host language program and process the program with the SQL precompiler.

  – You can create an SQL module file that contains SQL statements and compile the file with the SQL module processor. High-level language programs can link with the SQL module and call procedures from it to execute the SQL statements.

    See Chapter 3 for a description of the advantages of module language as compared with the SQL precompiler.

  Either way, with only minor adjustments, the statements you develop at the terminal using interactive SQL can be included in programs. Chapter 3 tells how to invoke the SQL module processor, and Chapter 4 tells how to invoke the SQL precompiler.

Section 1.4 describes the distinction SQL makes between executable and nonexecutable statements. Table 1–1 in Section 1.5 summarizes the SQL statements and those that are executable.

1.1.1 Invoking SQL Images

You can execute the RDB$SETVER.COM file to define global command language symbols for use with the Oracle Rdb precompilers and interactive utilities. This command procedure is particularly useful with the Oracle Rdb multiversion kit.

The following SQL symbols are defined and are valid in both the standard and multiversion environments:

- SQL$ == "SQL"

- SQL$PRE == "$SQL$PRE"
- SQL$MOD == "$SQL$MOD"

To define these symbols, type:

```
$ @SYS$LIBRARY:RDB$SETVER RESET
```

For more information regarding RDB$SETVER.COM and the corresponding RDB$SHOVER.COM, see the *Oracle Rdb Installation and Configuration Guide*.

1.2 Samples Directory

During installation, SQL installs a number of sample programs in a variety of languages in the Samples directory on your system. This directory also contains a command procedure or script to create sample databases.

The Samples directory is defined by the logical name SQL$SAMPLE.

1.3 How to Read Syntax Diagrams

This manual shows the format of SQL statements by using syntax diagrams. Syntax diagrams graphically portray optional, required, and repeating characteristics of SQL statements. You can learn the syntax of a statement by reading that statement's syntax diagram.

To read a syntax diagram, start at the upper-left corner and follow the arrows until you exit from the diagram at the bottom right corner. When you come to a branch in the path, choose the branch that contains the option you want. If you want to omit an option, choose the path with no language elements. If a diagram occupies more than one horizontal line, the arrow returns below the end of each line to the left margin. Syntax diagrams can contain:

- Names of other syntax diagrams

  If a diagram is named, the name appears in lowercase type above and to the left of the diagram. Syntax diagrams can refer to each other by name. The equal sign (=) indicates that the name is equivalent to the diagram and that the diagram can be substituted wherever that name appears in other diagrams. Such a substitution is similar to putting the name of a column where the syntax element column-name appears.

- Keywords

  Keywords appear in uppercase type. If a keyword is underlined, you *must* include it in the SQL statement. A keyword that is not underlined is optional. An optional keyword helps to make the statement more readable. Omitting or including an optional keyword has no effect on the statement.

- Punctuation marks

  Punctuation marks are included in the diagram when required by the syntax of the command or statement. For example, the semicolon (;) is a statement terminator in statements that require terminators.

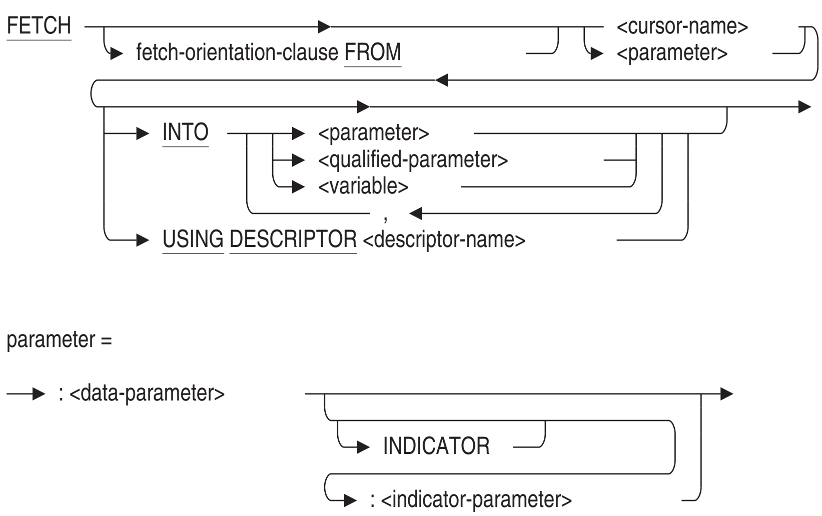
- User-supplied elements

  User-supplied elements appear in lowercase type and within angle brackets (< >) in syntax diagrams. These elements can include names, expressions, and literals. If a user-supplied element appears on the main line of a diagram, as cursor-name does in Figure 1–1, you are required to supply a substitute for the element.

  In text, a user-supplied element appears only in lowercase type.

Figure 1–1 shows the syntax diagram for the SQL FETCH statement.

Figure 1–1 Sample Syntax Diagram (FETCH)



Typically, the descriptions of SQL statements in Chapter 6 and Chapter 7 follow syntax diagrams. The description is presented as an argument list, with each entry of the list describing the corresponding element of the syntax diagram. The following list shows the format of such an argument list but describes syntax diagram conventions instead of the meaning of the arguments:

Arguments

FETCH
Is in uppercase type and underlined on the main line of the diagram. Therefore, you must supply this keyword.

cursor-name
parameter
Is in lowercase type in angle brackets on the main line of the diagram. Therefore, you must supply a substitute for cursor-name or parameter. In this manual, the description for user-supplied names such as cursor names and the description for parameters is part of Chapter 2. The argument list following a syntax diagram typically refers to Chapter 2 rather than repeating the description.

INTO
Is optional; however, if you chose that branch, you must supply this keyword.

parameter
qualified-parameter
variable
Is in lowercase type in angle brackets on a branch. Because it always parallels an empty branch, parameters and variables are optional. The subdiagram expands the definition of parameter.

comma
Is on a reverse loop. The loop indicates that you have the option to include more than one parameter or variable. If you do, they are separated by commas.

parameter
Is in lowercase type in angle brackets on a main branch. Parameters are optional, but if you include them, each one must contain a main parameter. An indicator parameter is optional and the keyword INDICATOR is optional.

All lowercase words are explained in the argument list that follows the diagram. Some explanations refer you to other diagrams that appear elsewhere in the manual.

1.4 Executable and Nonexecutable Statements

SQL distinguishes between executable and nonexecutable statements in host language programs, dynamic SQL, and interactive SQL.

- In host language programs, **nonexecutable** SQL statements are those that SQL processes completely when it precompiles a program or compiles an SQL module. **Executable** SQL statements also undergo processing during precompile time or module compile time but do not execute until the program runs.

  When embedded in host language programs or included in SQL modules, the following statements are nonexecutable:

  - BEGIN DECLARE (precompiled programs only)
  - DECLARE TRANSACTION
  - DECLARE ALIAS
  - DECLARE STATEMENT
  - DECLARE TABLE
  - DECLARE CURSOR
  - END DECLARE (precompiled programs only)
  - INCLUDE (precompiled programs only)
  - WHENEVER (precompiled programs only)

- In dynamic SQL, the following statement is nonexecutable:

  - DECLARE TRANSACTION

  Nonexecutable statements in dynamic SQL take effect when SQL processes the PREPARE statement for the statement. Issuing an EXECUTE statement for a nonexecutable statement in dynamic SQL is valid but does nothing.

- In interactive SQL, the following statements are nonexecutable:

  - DECLARE TRANSACTION
  - DECLARE CURSOR

  Nonexecutable statements in interactive SQL mean that the operation controlled by the statement does not occur until you enter an executable statement. For example, a transaction you define in a DECLARE TRANSACTION statement is not started until you enter a data manipulation or definition statement, such as SELECT. Similarly, the result table you define in a DECLARE CURSOR statement is not created until you enter an OPEN statement.

1.5 Summary of SQL Statements

Table 1–1 summarizes for all SQL statements the environments in which they are allowed and processed. Specifically, the table shows for each statement whether or not it can be:

- Issued interactively
- Embedded in host language programs to be precompiled
- Used as part of an SQL module language file
- Supplied to a program at run time for dynamic execution
- Treated as executable by SQL
- Included both in a simple and a compound statement (S/C), only in a simple statement (S), or only in a compound statement (C)

For more information about using a statement in a particular environment, including information about any restrictions, see Chapter 6, Chapter 7, and Chapter 8.

Table 1–1 Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
ACCEPT	X				X	S
ALTER CONSTRAINT	X	X	X	X	X	S
ALTER DATABASE	X	X	X	X	X	S
ALTER DOMAIN	X	X	X	X	X	S
ALTER FUNCTION	X	X	X	X	X	S
ALTER INDEX	X	X	X	X	X	S
ALTER MODULE	X	X	X	X	X	S
ALTER PROCEDURE	X	X	X	X	X	S
ALTER PROFILE	X	X	X	X	X	S
ALTER OUTLINE	X	X	X	X	X	S

(continued on next page)

Table 1–1 (Cont.) Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
ALTER ROLE	X	X	X	X	X	S
ALTER SEQUENCE	X	X	X	X	X	S
ALTER SYNONYM	X	X	X	X	X	S
ALTER STORAGE MAP	X	X	X	X	X	S
ALTER TABLE	X	X	X	X	X	S
ALTER TRIGGER	X	X	X	X	X	S
ALTER USER	X	X	X	X	X	S
ATTACH	X	X	X	X	X	S
BEGIN DECLARE		X				S
CALL	X	X	X	X	X	S/C
CASE	X	X	X	X	X	C
CLOSE	X	X	X		X	S
COMMENT ON	X	X	X	X	X	S
COMMIT	X	X	X	X	X	S/C
Compound statement	X	X	X	X	X	S/C
CONNECT	X	X	X	X	X	S
CREATE CATALOG	X	X	X	X	X	S
CREATE COLLATING SEQUENCE	X	X	X	X	X	S
CREATE DATABASE	X	X	X	X	X	S
CREATE DOMAIN	X	X	X	X	X	S
CREATE FUNCTION	X	X	X	X	X	S
CREATE INDEX	X	X	X	X	X	S
CREATE MODULE	X	X	X	X	X	S
CREATE OUTLINE	X	X	X	X	X	S
CREATE PROCEDURE	X	X	X	X	X	S
CREATE PROFILE	X	X	X	X	X	S
CREATE ROLE	X	X	X	X	X	S
CREATE SCHEMA	X	X	X	X	X	S

(continued on next page)

Table 1–1 (Cont.) Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
CREATE SEQUENCE	X	X	X	X	X	S
CREATE STORAGE MAP	X	X	X	X	X	S
CREATE SYNONYM	X	X	X	X	X	S
CREATE TABLE	X	X	X	X	X	S
CREATE TRIGGER	X	X	X	X	X	S
CREATE USER	X	X	X	X	X	S
CREATE VIEW	X	X	X	X	X	S
DECLARE ALIAS		X	X			S
DECLARE CURSOR	X	X	X			S
Dynamic DECLARE CURSOR		X	X			S
Extended Dynamic DECLARE CURSOR		X	X		X	S
DECLARE FUNCTION	X			X	X	S
DECLARE LOCAL TEMPORARY TABLE	X			X	X	S
DECLARE MODULE		X				S
DECLARE PROCEDURE	X			X	X	S
DECLARE STATEMENT		X	X			S
DECLARE TABLE		X	X			S
DECLARE variable	X				X	S/C
DECLARE TRANSACTION	X	X	X	X		S
DELETE	X	X	X	X	X	S/C
DESCRIBE		X	X		X	S
DISCONNECT	X	X	X	X	X	S
DROP CATALOG	X	X	X	X	X	S
DROP COLLATING SEQUENCE	X	X	X	X	X	S
DROP CONSTRAINT	X	X	X	X	X	S
DROP DATABASE	X	X	X	X	X	S

(continued on next page)

Table 1–1 (Cont.) Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
DROP DOMAIN	X	X	X	X	X	S
DROP FUNCTION	X	X	X	X	X	S
DROP INDEX	X	X	X	X	X	S
DROP MODULE	X	X	X	X	X	S
DROP OUTLINE	X	X	X	X	X	S
DROP PATHNAME	X	X	X	X	X	S
DROP PROCEDURE	X	X	X	X	X	S
DROP ROLE	X	X	X	X	X	S
DROP SCHEMA	X	X	X	X	X	S
DROP SEQUENCE	X	X	X	X	X	S
DROP STORAGE MAP	X	X	X	X	X	S
DROP SYNONYM	X	X	X	X	X	S
DROP TABLE	X	X	X	X	X	S
DROP TRIGGER	X	X	X	X	X	S
DROP USER	X	X	X	X	X	S
DROP VIEW	X	X	X	X	X	S
EDIT	X				X	
END DECLARE		X				S
Execute (@)	X				X	
EXECUTE		X	X		X	S
EXECUTE IMMEDIATE		X	X		X	S
EXIT	X				X	
EXPORT	X				X	
FETCH	X	X	X		X	S
FOR	X	X	X	X	X	C
GET DIAGNOSTICS	X	X	X	X	X	C
GET ENVIRONMENT	X				X	S

(continued on next page)

Table 1–1 (Cont.) Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
GRANT	X	X	X	X	X	S
GRANT (ANSI style)	X	X	X	X	X	S
HELP	X				X	
IF	X	X	X	X	X	C
IMPORT	X				X	
INCLUDE		X				S
INSERT	X	X	X	X	X	S/C
INTEGRATE	X				X	
ITERATE	X	X	X	X	X	C
LOCK TABLE	X	X	X	X	X	S/C
LEAVE	X	X	X	X	X	C
LOOP	X	X	X	X	X	C
OPEN	X	X	X		X	S
Operating System Invocation ($)	X				X	
PREPARE		X	X		X	S
PRINT	X				X	
QUIT	X				X	
RELEASE		X	X		X	S
RENAME	X	X	X	X	X	S
REPEAT	X	X	X	X	X	C
RETURN	X	X	X	X	X	C
REVOKE	X	X	X	X	X	S
REVOKE (ANSI style)	X	X	X	X	X	S
ROLLBACK	X	X	X	X	X	S/C
SELECT (general form)	X			X	X	S
SELECT (singleton select)	X	X	X	X	X	S/C
SET	X				X	C

(continued on next page)

Table 1–1 (Cont.) Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
SET ALIAS	X	X	X	X	X	S
SET ALL CONSTRAINTS	X	X	X	X	X	S
SET assignment	X	X	X	X	X	C
SET AUTOMATIC TRANSLATION	X				X	S
SET CATALOG	X	X	X	X	X	S
SET CHARACTER LENGTH	X	X	X	X	X	S
SET COMPOUND TRANSACTIONS	X	X	X	X	X	S
SET CONNECT	X	X	X	X	X	S
SET DEFAULT CHARACTER SET	X	X	X	X	X	S
SET DEFAULT DATE FORMAT	X	X	X	X	X	S
SET DIALECT	X	X	X	X	X	S
SET DISPLAY	X					S
SET DISPLAY CHARACTER SET	X	X	X	X	X	S
SET ECHO	X				X	S
SET FEEDBACK	X				X	S
SET FLAGS	X			X	X	S
SET HEADING	X				X	S
SET HOLD CURSORS	X	X	X	X	X	S
SET IDENTIFIER CHARACTER SET	X	X	X	X	X	S
SET KEYWORD RULES	X	X	X	X	X	S
SET LITERAL CHARACTER SET	X	X	X	X	X	S
SET NAMES	X	X	X	X	X	S
SET NATIONAL CHARACTER SET	X	X	X	X	X	S
SET NULL	X				X	S
SET OPTIMIZATION LEVEL	X	X	X	X	X	S
SET QUIET COMMIT	X				X	S
SET QUOTING RULES	X	X	X	X	X	S

(continued on next page)

Table 1–1 (Cont.) Summary of SQL Statements

Statement	Inter-active	Pre-compiled	Module Language	Dynamically Executable	Executable	Simple and/or Compound
SET SCHEMA	X	X	X	X	X	S
SET SESSION AUTHORIZATION	X	X	X	X	X	S
SET TRANSACTION	X	X	X	X	X	S/C
SET VIEW UPDATE RULES	X	X	X	X	X	S
SHOW	X				X	
SIGNAL	X	X	X	X	X	C
START TRANSACTION	X	X	X	X	X	S/C
TRACE	X	X	X	X	X	C
TRUNCATE TABLE	X	X	X	X	X	S
UNDECLARE variable	X				X	S
UPDATE	X	X	X	X	X	S
WHENEVER		X				S
WHILE	X	X	X	X	X	S

1.6 Keywords and Line Terminators

In syntax diagrams, keywords are shown in uppercase type. In interactive SQL, you can abbreviate keywords as long as the abbreviation uniquely specifies a syntactically allowed choice. You cannot abbreviate keywords in SQL statements used in a host language program (precompiled, module language, or dynamic SQL).

There are two types of keywords:

- Required - uppercase and underlined
- Optional - uppercase only

1.6.1 Statement Terminators and Comment Characters

You must end SQL statements in different ways depending on the environment in which you issue them:

- In interactive SQL, you must end statements with a semicolon (;).

  The only exceptions to this rule are statements that are valid only within interactive SQL: the operating system invocation ($), Execute (@), EDIT, EXIT, QUIT, SET, and SHOW.

You can explicitly continue a line in interactive SQL by ending it with a hyphen (-). The continuation character takes precedence over the minus operator. If you intend to use the hyphen as the minus operator in a query, avoid typing it as the last element on a continued line. You can, however, type a double hyphen as shown below:

```
SQL> SELECT col1 - -
cont> col2 FROM my_table;
```

In the preceding example, the second hyphen is interpreted as the continuation character leaving the first hyphen to be interpreted as the minus operator.

You can also use the SET with the CONTINUE CHARACTER argument to define the continuation character for interactive SQL. See the SET Statement for more information.

- In precompiled programs, the statement terminator depends on the host language:
  - COBOL: END-EXEC
  - FORTRAN: none required
  - Ada: a semicolon (;)
  - C: a semicolon (;)
  - Pascal: a semicolon (;)
  - PL/I: a semicolon (;)
- In SQL module language files, DECLARE statements are not terminated. Other statements end in a semicolon.
- Dynamic SQL allows an optional statement terminator. In some cases, this statement terminator will assist SQL in processing ambiguous syntax.

As with statement terminators, the notation that SQL recognizes as denoting comments depends on the environment:

- Interactive SQL interprets the exclamation point (!) or the double hyphen (--) as a comment character. Interactive SQL disregards any characters on a line following an exclamation point or double hyphen.

Note

Oracle Rdb recommends use of the double hyphen (--) as a comment character. This allows portability of interactive SQL statements into

SQL module language programs and is also in conformance with the ANSI/ISO standard.

- Oracle Rdb supports multiline and embedded comments for compatibility with the Oracle RDBMS.

  These comments start with the characters "/*" and are terminated (possibly on a subsequent line) by "*/". There can be no spaces between the "*" and "/" characters which open and close a comment.

  Because this style of comment is often used to disable statements in an SQL command scripts, Rdb supports nested comments. This allows comments within an existing sequence of SQL commands to be ignored when used in larger comment sequence. i.e. only the matching */ will close the comment. This is shown in the following examples.

  Note

  Nested comments are not supported by the ORACLE LEVEL1 or ORACLE LEVEL2 dialects.

  Because an unterminated comment is often difficult to find, the interactive SQL prompt is changed to '***>' for each line which is a comment continuation.

  Oracle Rdb continues to support the exclamation point (!) and the double hyphen (--) as comment introducers which can be used to comment the end of a single line. The end-of-line is considered the comment closer.

  Example:

  The following examples show embedded and a multiline comments.

```
SQL> -- show that text is hidden
SQL> show /* current */ version
Current version of SQL is: Oracle Rdb SQL V8.0-00
SQL>
SQL> /*
***> **********************************
***> The following query fetches the
***> JOB_HISTORY row for the current job
***> of employee 164 (aka Alvin Toliver)
***> ***********************************
***> */
SQL> select department_code,
cont>        job_start /*, employee_id*/
cont> from JOB_HISTORY
cont> where employee_id = '00164'
cont>   and job_end is null;
 DEPARTMENT_CODE   JOB_START
 MBMN              21-Sep-1981
1 row selected
SQL>
SQL> /* disable this CREATE VIEW
***> create view V_00164
***>     as select e.last_name, /* e.middle_initial, */ e.first_name,
***>          sh.salary_amount
***>    from EMPLOYEES e, SALARY_HISTORY sh
***>    where e.employee_id = '00164'
***>          and sh.employee_id = e.employee_id;
***>
***> select * from V_00164;
***> select count/*all*/(*) from V_00164;
***> */
SQL>
```

- The SQL precompiler uses host language rules for comments in embedded SQL statements.
- The SQL module processor interprets the double hyphen as a comment character.
- Dynamic SQL interprets the double hyphen as a comment character.

1.7 Support for Multivendor Integration Architecture

Oracle Rdb supports **Multivendor Integration Architecture (MIA)**. This support includes the following:

- Several character sets in addition to the DEC Multinational Character Set (MCS). For information about the supported character sets, see Section 2.1.
- Using multiple character sets in one database.

- Specifying character sets for database objects, identifiers, literals, and character data type parameters.
- Using delimited identifiers to maintain the case of an identifier. For more information, see Section 2.2.
- Specifying character lengths and offsets in characters, rather than octets.
- Using the SET DIALECT 'MIA' statement that sets the MIA-compliant character sets, which are:
  – Default character set: KATAKANA
  – National character set: KANJI
  – Identifier character set: DEC_KANJI
  – Literal character set: KATAKANA

In addition, SQL provides support for new character data type variables in SQL precompiled C, COBOL, and FORTRAN programs. See the description of the data types in Sections 4.4.3, 4.4.4, and 4.4.5.

SQL also lets you specify a particular character set or the national character set for formal parameters in SQL module language. For more information, see Section 3.2 and DECLARE MODULE Statement.

When you use character sets other than MCS, be aware of the effect the character set has on the following elements:

- LIKE predicate: See Section 2.7.8.
- Substrings: See Section 2.6.2.16.
- Conversions between data types: See Section 2.3.8.
- String concatenation operator: See Section 2.6.6.

During installation, SQL installs a number of sample programs in a variety of languages. These programs are in the Samples directory on your system, and a brief description of each can be found in the file about_sql_examples.txt in the samples directory. One of these programs, called MIA_CHAR_SET, creates a database that uses the different character sets required to be MIA compliant:

- sql_mia_char_set_mod.c is the C program source file.
- sql_mia_char_set_c.sqlmod is the SQL module file.
- sql_mia_char_set_pre.sc is the SQL precompiler file.

2

Language and Syntax Elements

SQL uses a number of basic syntax and language elements that are common to many statements. These elements are sometimes referred to in syntax diagrams without further explanation. This chapter describes those elements:

- Character sets
- Names
- Data types
- Literals
- SQL and DATATRIEVE formatting clauses
- Value expressions
- Predicates
- Select expressions and column select expressions

For guidelines on how to form SQL statements to perform certain tasks, see the *Oracle Rdb Introduction to SQL* and the *Oracle Rdb Guide to SQL Programming*.

2.1 Supported Character Sets

Oracle Rdb supports multiple character sets and lets you use more than one character set in a database.

Table 2–1 shows the supported character sets, their names as you specify them in SQL statements, and descriptions of the character sets.

Table 2–1 Supported Character Sets

Character Set[1]	Description
AL24UTFFSS	UTF-8 encoding based on the Unicode standard 1.1
ASCII	7-bit ASCII characters
ARABIC	Arabic characters as defined by the ISO9036 standards
BIG5	A set of characters used by the Taiwan information industry
DEC_HANYU	Traditional Chinese characters (Hanyu) as used in Taiwan and defined by standard CNS11643:1986, supplemental characters as defined by DTSCC and ASCII
DEC_HANZI	Chinese (Bopomofo) characters as defined by standard GB2312:1980 and ASCII characters
DEC_KANJI	Japanese characters as defined by the JIS X0208:1990 standard, Hankaku Katakana characters as defined by JIS X0201:1976 prefixed by SS2 (8E hex), user-defined characters, and ASCII characters
DEC_KOREAN	Korean characters as defined by standard KS C5601:1987 and ASCII characters
DEC_MCS	A set of international alphanumeric characters, including characters with diacritical marks
DEC_SICGCC	Traditional Chinese characters (Hanyu) as used in Taiwan and defined by standard CNS11643:1986 and ASCII
DEVANAGARI	Devanagari characters as defined by the ISCII:1988 standard
DOS_LATIN1	DOS Latin 1 code
DOS_LATINUS	DOS Latin US code
GB18030	Simplified Chinese characters as used by the People's Republic of China (PRC) and defined by the GB18030-2000 standard
HANYU	Traditional Chinese characters (Hanyu) as used in Taiwan and defined by the standard CNS11643:1986
HANZI	Chinese (Bopomofo) characters as defined by standard GB2312:1980
HEX	Translation of text data to and from hexadecimal data
ISOLATIN1	Extended European characters as defined by the ISO/IEC 8859-1:1987 standard
ISOLATIN9	Extended European characters as defined by the ISO/IEC 8859-15 standard which includes the Euro Character code-point.
ISOLATINARABIC	Arabic characters as defined by the ISO/IEC 8859-6:1987 standard

[1]To allow easy portability of applications across national boundaries, you can use a logical name in place of a character set name. See Section 2.1.11 for more information.

(continued on next page)

Table 2–1 (Cont.) Supported Character Sets

Character Set[1]	Description
ISOLATINCYRILLIC	Cyrillic characters as defined by the ISO/IEC 8859-5:1987 standard
ISOLATINGREEK	Greek characters as defined by the ISO/IEC 8859-7:1987 standard
ISOLATINHEBREW	Hebrew characters as defined by the ISO/IEC 8859-8:1987 standard
KANJI	Japanese characters as defined by the JIS X0208:1990 standard and user-defined characters
KATAKANA	Japanese phonetic alphabet (Hankaku Katakana), as defined by standard JIS X0201:1976
KOREAN	Korean characters as defined by standard KS C5601:1987
SHIFT_JIS	Japanese characters as defined by the JIS X0208:1990 standard using Shift_JIS specific encoding scheme, Hankaku Katakana characters as defined by JIS X0201:1976, and ASCII characters
TACTIS	Thai characters based on TACTIS (Thai API Consortium/Thai Industrial Standard) which is a combination of ISO 646-1983 and TIS 620-2533 standards
UNICODE	Unicode characters as described by Unicode Standard and ISO/IEC 10646 transformation format UTF-16
UNSPECIFIED	See Section 2.1.10.
UTF8	Unicode characters as described by Unicode Standard and ISO/IEC 10646 UTF-encoding form
WIN_ARABIC	MS Windows Code Page 1256 8-Bit Latin/Arabic
WIN_CYRILLIC	MS Windows Code Page 1251 8-Bit Latin/Cyrillic
WIN_GREEK	MS Windows Code Page 1253 8-Bit Latin/Greek
WIN_HEBREW	MS Windows Code Page 1255 8-Bit Latin/Hebrew
WIN_LATIN1	MS Windows Code Page 1252 8-Bit West European

[1]To allow easy portability of applications across national boundaries, you can use a logical name in place of a character set name. See Section 2.1.11 for more information.

Any of the supported character sets can be used whenever character sets are specified, except as the identifier character set. For information about the identifier character set, see Section 2.1.5.

Character sets differ in how characters are coded. That is, characters in some character sets are coded entirely in one octet; characters in other character sets are coded in more than one octet. (An **octet** is a group of 8 bits.)

The various ways characters can be coded are:

- Single-octet

  A **single-octet character set** is entirely represented in one octet. ASCII is an example of a single-octet character set. Each ASCII character is represented in one octet.

- Multi-octet

  A **multi-octet character set** is, in general, entirely represented in one or more octets. Some character sets are fixed multi-octet character sets and some are mixed multi-octet characters.

  – Fixed multi-octet

    A **fixed multi-octet character set** is represented by two or more fixed number of octets. Kanji is an example of a fixed multi-octet character set. Each Kanji character is represented in two octets.

  – Mixed multi-octet

    A **mixed multi-octet character set** is represented by one or more mixed number of octets that allow the use of ASCII and a fixed multi-octet character set in the same string. DEC_KANJI is an example of a mixed multi-octet character set. The ASCII characters are represented in one octet, and the Kanji characters are represented in two octets.

Table 2–2 shows how many octets each of the supported character sets uses to code a single character.

Table 2–2 Number of Octets Used by Characters in Character Sets

Character Set	Number of Octets Used for Each Character
Single-Octet Character Sets	
ASCII	One octet
DEC_MCS	One octet
DOS_LATIN1	One octet
DOS_LATINUS	One octet
ISOLATINARABIC	One octet
ISOLATINHEBREW	One octet
ISOLATINCYRILLIC	One octet

(continued on next page)

Table 2–2 (Cont.) Number of Octets Used by Characters in Character Sets

Character Set	Number of Octets Used for Each Character
Single-Octet Character Sets	
ISOLATINGREEK	One octet
ISOLATIN1	One octet
ISOLATIN9	One octet
DEVANAGARI	One octet
KATAKANA	One octet
TACTIS	One octet
UNSPECIFIED	One octet
WIN_*	One octet
Fixed Multi-Octet Character Sets	
BIG5	Two octets
HEX	Two octets
HANYU	Two octets
HANZI	Two octets
KANJI	Two octets
KOREAN	Two octets
UNICODE	Two octets
Mixed Multi-Octet Character Sets	
AL24UTFFSS	One to three octets as specfied by UTF-8 standard
DEC_KOREAN	One octet for ASCII characters; two octets for KOREAN characters
DEC_SICGCC	One octet for ASCII characters; two octets for Hanyu characters

(continued on next page)

Table 2–2 (Cont.) Number of Octets Used by Characters in Character Sets

Character Set	Number of Octets Used for Each Character
Mixed Multi-Octet Character Sets	
DEC_HANYU	One octet for ASCII characters; two octets for Hanyu characters; four octets for supplemental characters
DEC_HANZI	One octet for ASCII characters; two octets for HANZI characters
DEC_KANJI	One octet for ASCII characters; two octets for KANJI characters; two octets for Hankaku Katakana characters (SS2 (8E hex) prefix plus single octet JIS X0201 character)
SHIFT_JIS	One octet for ASCII characters; one octet for Hankaku Katakana characters; two octets for KANJI characters using SHIFT_JIS encoding
UTF8	One to three octets as specfied by UTF-8 standard

You cannot use a multi-octet character in an edit string or in a file name, repository path name, or database name.

2.1.1 Automatic Translation

During operations on text data such as assignments of a literal to a text column or the comparison of two string variables, Oracle Rdb carries out character set compatibility checks to ensure that the operation is viable.

Without automatic translation being enabled this checking is quite restrictive in that in most cases the two text objects must have identical character set before the operation is allowed.

The automatic translation feature allows you to choose whether the character set checking should be restrictive or whether Rdb should attempt a character set translation, similar to that provided by the TRANSLATE function prior to assignments or comparisons.

With automatic translation enabled you may easily carry out operations that previously required additional translations steps to be carried out explicitly.

1. Carry out comparisons between columns that contain data encoded in different character sets that have common character subsets, for example, DEC_MCS and DEC_KANJI have ASCII in common.

2. Use the same SQL code to access database data irrespective of the client's environment. For example, a user on a Japanese PC accessing a DEC_MCS column would have to add TRANSLATE statements to the SQL commands to convert the DEC_MCS to SHIFT_JIS before they could display it on their screen. With automatic translation enabled and a Display Character set specified, this would not be required.
3. Enter data from a native interface without explicit translations. For example, users using SHIFT_JIS on a Japanenese PC may access and insert data into a DE_KANJI column in the database without explicit translations statements.

You may enabled automatic translation by:

1. Using a SET AUTOMATIC TRANSLATION statement
2. Defining the SQL$AUTOMATIC_TRANSLATION logical name

SQL$AUTOMATIC_TRANSLATION

The logical name SQL$AUTOMATIC_TRANSLATION allows SQL users to specify that automatic translations should be enabled by default.

The logical SQL$AUTOMATIC_TRANSLATION may be placed in any logical name table accessible to the client SQL process.

If the logical name is set to either the string 'TRUE' or the value 'T' prior to invoking SQL, then automatic translation will be enabled by default, any other value will disable automatic translation within SQL.

2.1.2 Character Set HEX

The character set HEX is comprised of two octet hexadecimal characters '00' through 'FF' and has the characteristic that the contents of data objects with this character set will not be automatically translated to the display character set when automatic translation has been enabled.

It may be used in conjunction with the CAST and TRANSLATE functions to obtain the hexadecimal equivalence of text objects.

Translation to the HEX character set will translate source data octet by octet into hexadecimal notation.

Translation from the HEX character set will translate from hexadecimal notation to the destination character set.

For example:

```
SQL> show character sets
Default character set is DOS_LATINUS
National character set is DOS_LATINUS
Identifier character set is DOS_LATINUS
Literal character set is DOS_LATINUS
Display character set is DOS_LATINUS

Alias RDB$DBHANDLE:
 Identifier character set is DEC_MCS
 Default character set is DEC_MCS
 National character set is DEC_MCS
SQL> show automatic translation
Automatic translation: ON
SQL> create tab latin (f1 char(4) char set win_latin1,
cont> f2 char(4) char set dos_latinus);
SQL> insert into latin value ('AÉÖ','AÉÖ');
1 row inserted
SQL> select f1, cast(f1 as char(8) char set hex),
cont> f2, cast(f2 as char(8) char set hex) from latin;
 F1     F2
 AÉÖ 41C9D620   AÉÖ   41909920
1 row selected
SQL> select cast (_hex'9099' as char(2) ) from rdb$database;

 ÉÖ
1 row selected
SQL> select translate (_hex'9099' using rdb$dos_latinus )
Cont> from rdb$database;
  ÉÖ
1 row selected
```

The previous example also shows automatic translations between the literals character set DOS_LATINUS and the field F2 containing WIN_LATIN1, and the subsequent automatic translation from the F2 field back to the display character set.

The hexadecimal display of the field contents shows that the actual data stored in the database is different for field f1 and f2 even though the input literals and the output displayed appears identical.

2.1.3 Default Character Sets

The **default character set** is the character set that SQL uses for the following elements:

- Database columns with a character data type that does not explicitly specify a character set
- Parameters that are not qualified by a character set

You can specify the default character set at the session and database level. See the *Oracle Rdb Introduction to SQL* and *Oracle Rdb Guide to Database Design and Definition* for more detail about session and database character sets.

You can specify the database default character set only when you create the database. You cannot change the database default character set after you have created the database.

SQL uses DEC_MCS as the default character set, unless you have set the dialect to MIA or specified a default character set at the session level. You can override any default character set by specifying another default character set when creating a database.

To specify the default character set, use one of the character set names listed in Section 2.1.

The default character set does not affect the setting of the currency sign.

When you compile SQL programs (either SQL module language or precompiled SQL), SQL uses the following to derive the default character set:

- The DEFAULT CHARACTER SET clause in the DECLARE ALIAS statement specifies the default character set of the alias at *compile time*. At *run time*, SQL uses the default character set of the attached database. At run time, you must ensure that the database default character set is identical to the default character set specified in the DECLARE ALIAS clause.
- The DEFAULT CHARACTER SET clause of the SQL module header or the DECLARE MODULE statement specifies the character set for parameters that are not qualified by a character set.
- In dynamic SQL, the SET DEFAULT CHARACTER SET statement specifies, at run time, the character set for parameters that are not qualified by a character set.
- The RDB$CHARACTER_SET logical name. However, the logical name is deprecated and will not be supported in a future release.

2.1.4 Display Character Set

The display character set is the character set SQL uses for determining the character set that text will automatically be translated to before display in interactive SQL or for text being returned by SQL to a user program.

You can specify the display character set only for a session or a module by using the SET DISPLAY CHARACTER SET statement or the DISPLAY CHARACTER SET clause of the SQL module header, the DECLARE MODULE statement, or the DECLARE ALIAS statement.

The choice of display character set is limited to those character sets that include ASCII characters. Section 2.1.5 identifies the subset of character sets that you can use to specify the display character set.

2.1.5 Identifier Character Set

The **identifier character set** is the character set SQL uses for database object names, such as table names and column names. You can specify the identifier character set at the session and database level. The choice of identifier character set is limited to those character sets that include ASCII characters. This is necessary so that the object names for the Oracle Rdb system metadata, which is in ASCII, can be stored.

You can specify the identifier character set for the database *only* when you create the database. You cannot alter the identifier character set of a database after creation.

Following is a list of the character sets used for the identifier character set:

- ASCII
- AL24UTFFSS
- DEC_MCS
- DOS_LATIN1
- DOS_LATINUS
- DEVANAGARI
- DEC_SICGCC
- DEC_HANYU
- DEC_HANZI
- ISOLATINARABIC
- ISOLATINCYRILLIC
- ISOLATIN1
- ISOLATIN9
- ISOLATINGREEK
- ISOLATINHEBREW
- DEC_KANJI
- KATAKANA
- DEC_KOREAN

- SHIFT_JIS
- UTF8
- UNSPECIFIED
- TACTIS
- WIN_ARABIC
- WIN_GREEK
- WIN_CYRILLIC
- WIN_HEBREW

When you compile SQL programs (either SQL module language or precompiled SQL), SQL uses the following to derive the identifier character set:

- The IDENTIFIER CHARACTER SET clause of the SQL module header or the DECLARE MODULE statement specifies the character set for parameters that are not qualified by a character set.
- In dynamic SQL, the SET IDENTIFIER CHARACTER SET statement specifies, at run time, the character set for parameters that are not qualified by a character set.
- The RDB$CHARACTER_SET logical name. However, the logical name is deprecated and will not be supported in a future release.

SQL uses DEC_MCS as the identifier character set, unless you have set the dialect to MIA or specified an identifier character set at the session level. You can override any identifier character set by specifying another identifier character set when creating a database.

2.1.6 Literal Character Sets

The **literal character set** is the character set SQL uses for unqualified character string literals.

You can specify the literal character set only for a session or a module by using the SET LITERAL CHARACTER SET statement or the LITERAL CHARACTER SET clause of the SQL module header, the DECLARE MODULE statement, or the DECLARE ALIAS statement.

When inserting data into a column, you must qualify the literal with the same character set with which you defined the column.

For example, suppose that the literal character set of the module is DEC_MCS. If the column ENGLISH is defined as data type DEC_MCS, SQL returns an error when you execute the following statement:

```
SQL> INSERT INTO COLOURS
cont>    (ENGLISH)
cont> VALUES
cont>    (_DEC_KANJI'Black');
%SQL-F-INCCSASS, Incompatible character set assignment between ENGLISH and
<value expression>
SQL>
```

2.1.7 National Character Set

The **national character set** is a shorthand notation that you can use for a character set of your choice. SQL uses the national character set for the following elements:

- For all columns and domains with the data type NCHAR or NCHAR VARYING and for the NCHAR data type in a CAST function. For information about these data types, see Section 2.3.1.
- For all parameters in SQL module language with the data type NCHAR or NCHAR VARYING.
- For all character string literals qualified by the national character set; that is, the literal is preceded by the letter N and a single quotation mark (for example, N'). For more information, see Section 2.4.2.1.2.

To specify the national character set, use one of the character set names listed in Table 2–1.

You can specify the national character set at the session and database level. See the *Oracle Rdb Introduction to SQL* and the *Oracle Rdb Guide to Database Design and Definition* for more detail about session and database character sets.

You specify the national character set for a database when you create the database. You cannot alter the national character set of a database.

SQL uses DEC_MCS as the national character set, unless you have set the dialect to MIA or specified a national character set at the session level. You can override any national character set by specifying another national character set when creating a database.

When you compile SQL programs (either SQL module language or precompiled SQL), SQL uses the following to derive the national character set:

- The NATIONAL CHARACTER SET clause in the DECLARE ALIAS statement specifies the national character set of the alias at compile time.

It controls the national character set for column and domain definitions and the NCHAR and NCHAR VARYING data types in a CAST function. At run time, SQL uses the national character set of the attached database for these elements.

- The NATIONAL CHARACTER SET clause of the SQL module header and the DECLARE MODULE statement specifies the character set for literals qualified by the national character set and for parameters defined with the data type NCHAR or NCHAR VARYING.
- In dynamic SQL, the SET NATIONAL CHARACTER SET statement specifies, at run time, the character set for columns with the data type NCHAR and NCHAR VARYING and for character string literals qualified by the national character set.
- The RDB$CHARACTER_SET logical name. However, the logical name is deprecated and will not be supported in a future release.

Note

Although SQL does not require that the national character set of the database and the module match, Oracle Rdb recommends that you define both with the same character set.

2.1.8 Character Set ISOLATIN9

Oracle Rdb supports the ISOLATIN9 character set (as described by ISO 8859-15). ISOLATIN9 is similar to ISOLATIN1 except for 8 codepoints.

Table 2–3 compares ISOLATIN9 and ISOLATIN1.

Table 2–3 ISOLATIN1/ISOLATIN9 Character Set Differences

ISO Latin 1			ISO Latin 9	
Code Pos Hex	Unicode Pos Hex	Name	Unicode Pos Hex	Name
A4	00A4	currency symbol	20AC	euro sign
A6	00A6	broken bar	0160	latin capital letter s with caron
A8	00A8	diaeresis	0161	latin small letter s with caron
B4	00B4	acute accent	017D	latin capital letter z with caron
B8	00B8	cedilla	017E	latin small letter z with caron
BC	00BC	vulgar fraction one quarter	0152	latin capital ligature oe
BD	00BD	vulgar fraction one half	0153	latin small ligature oe
BE	00BE	vulgar fraction three quarters	0178	latin capital letter y with diaeresis

2.1.9 Oracle NLS Character Set Names

Oracle Rdb supports the use of Oracle National Language Support (NLS) names as aliases for existing Oracle Rdb character sets, as summarized in Table 2–4. You can use NLS alias names anywhere a character set name can be used.

Table 2–4 Oracle NLS Character Set Names Supported as Aliases

US7ASCII	ASCII
WE8DEC	DEC_MCS
WE8ISO8859P1	ISOLATIN1
WE8ISO8859P1	ISOLATIN9
CL8ISO8859P5	ISOLATINCYRILLIC
AR8ISO8859P6	ISOLATINARABIC
EL8ISO8859P7	ISOLATINGREEK
IW8ISO8859P8	ISOLATINHEBREW
TH8TISASCII	TACTIS
JA16VMS	DEC_KANJI
JA16SJIS	SHIFT_JIS
KO16KSC5601	KOREAN
ZHS16CGB231280	HANZI
ZH16BIG5	BIG5
JA16EUCFIXED	KANJI

2.1.10 Character Set UNSPECIFIED

Oracle Rdb supports the use of the UNSPECIFIED character set. You can make comparisons and assignments between text objects (columns, literals, and so on) that have the UNSPECIFIED character set, and any other text object regardless of the character set of the other text object.

The characteristics of the UNSPECIFIED character set are as follows:

- The character set ID is 32767.
- It can be used to specify any session or database character set, including the identifier character set.
- It is a single-octet character set (fixed).
- It applies casing (uppercase and lowercase) only to ASCII characters.
- It contains ASCII, as follows:
  - The space character is the ASCII space character (0x20).
  - The wildcard character is the ASCII underscore (0x5f).
  - The string wildcard is the ASCII percent (0x25).

2.1.11 Logical Names for Character Sets

You can define a logical name for a character set name. Doing so allows easy portability of applications across national boundaries. You can use this logical name or parameter anywhere you use a character set name in SQL. SQL translates the logical name or parameter at compile time for precompiled SQL and SQL module language, or at run time for dynamic SQL and interactive SQL.

The logical name can begin with any of the following:

- RDBVMS$
- RDB$
- SQL$

  Oracle Rdb recommends that you begin logical names with RDB$.

The logical name or parameter must translate to a valid character set name found in Table 2–1.

The following example shows how to define and use a logical name for a character set:

```
$ DEFINE RDB$LOCAL_CHAR_SET KANJI
$ SQL
SQL> ATTACH 'FILENAME personnel';
SQL> CREATE DOMAIN SURNAME_DOM CHAR(20) CHARACTER SET RDB$LOCAL_CHAR_SET;
SQL> SHOW DOMAIN SURNAME_DOM
SURNAME_DOM                     CHAR(20)
         RDB$LOCAL_CHAR_SET 10 Characters,  20 Octets
SQL>
```

2.2 User-Supplied Names

You must supply names (identifiers) to satisfy the syntax of SQL statements that require user-supplied names. In statement syntax diagrams, user-supplied names are shown in lowercase type.

User-supplied names must:

- Be no more than 31 octets (8-bit characters).
- Conform to one of the following rules:
  - If the identifier character set is MCS, the name must contain only alphanumeric characters and begin with an uppercase or lowercase letter. **Alphanumeric characters** are uppercase or lowercase letters

(A, a), including letters with diacritical marks (À), digits, dollar signs ($), and underscores (_).

Uppercase and lowercase letters are treated equally.

Although dollar signs are valid characters in names, to avoid conflicts it is recommended that you do not use them.

You cannot begin a user-supplied name with a numeric character.

– If the identifier character set is other than DEC_MCS, it can contain only a valid sequence of characters as defined by the standard for that character set. See Section 2.1 for information about the standards for each character set.

– The name can be a delimited identifier. A **delimited identifier** is a user-supplied name enclosed in double quotation marks ("). It can start with and contain alphanumeric characters, special characters, control characters, and spaces. The quotation mark (") character can be included in a delimited identifier by typing two quotation marks together (see the following example). Trailing spaces are not significant. See Section 2.4.2.1 for a list of special characters. The alphabetic characters can be uppercase or lowercase. The following example shows many variations of delimited identifiers:

```
SQL> SET QUOTING RULES 'SQL99
SQL> CREATE TABLE "This is a Test"
cont> (""""              CHAR(5),
cont>  "_NAME"           CHAR(20),
cont>  "  City"          CHAR(20),
cont>  "1st_date"        DATE,
cont>  "A ""B and C"""   CHAR(10),
cont>  "$_Amount"        INT,
cont>  """Test"          CHAR(5)
cont> );
SQL> SHOW TABLE (COLUMN) "This is a Test";
Information for table This is a Test

Columns for table This is a Test:
Column Name                     Data Type        Domain
-----------                     ---------        ------
"                               CHAR(5)
_NAME                           CHAR(20)
  City                          CHAR(20)
1st_date                        DATE VMS
A "B and C"                     CHAR(10)
$_Amount                        INTEGER
"Test                           CHAR(5)
```

If you use a delimited identifier, SQL maintains the case of the identifier. That is, if you enclose the identifier Employee_ID in quotation marks ("Employee_ID"), SQL preserves the uppercase and lowercase letters. The delimited identifiers "Employee_ID", "EMPLOYEE_ID", and "employee_id" are distinct from each other.

You must enable ANSI/ISO SQL quoting when using delimited identifiers. You can enable ANSI/ISO SQL quoting in the following ways:

- Use the SET QUOTING RULES and SET DIALECT 'SQL99' statements in interactive SQL.
- Use the QUOTING RULES clause in the SQL module file to enable ANSI/ISO SQL quoting for the compilation.
- Use the QUOTING RULES clause in a DECLARE MODULE statement embedded in a program to be precompiled.

If you want to use a keyword as a user-supplied name, you must set the quoting rules or dialect to SQL99 and use the delimited identifier. For example:

```
SQL> SET DIALECT 'SQL99';
SQL> -- You must use the delimited identifier to create
SQL> -- a domain named DATE.  If you do not, SQL returns an
SQL> -- error message.
SQL> --
SQL> CREATE DOMAIN DATE CHAR (100);
%SQL-F-RES_WORD_AS_IDE, Keyword DATE used as an identifier
SQL> --
SQL> CREATE DOMAIN "DATE" CHAR (100);
SQL> SHOW DOMAIN "DATE"
DATE                            CHAR(100)
SQL> --
SQL> -- You must also use the delimited identifier around
SQL> -- the user-supplied table name if you want to use the domain
SQL> -- DATE; otherwise, the data type DATE will be referenced.
SQL> --
SQL> CREATE TABLE ABC
cont> (FIELD_1 "DATE",
cont>  FIELD_2 "DATE",
cont>  FIELD_3 DATE);
SQL> --
SQL> SHOW TABLE (COLUMNS) ABC;
Information for table ABC
```

```
Columns for table ABC:
Column Name                     Data Type        Domain
-----------                     ---------        ------
FIELD_1                         CHAR(100)        DATE
FIELD_2                         CHAR(100)        DATE
FIELD_3                         DATE ANSI
```

See the SET DIALECT Statement and the *Oracle Rdb Introduction to SQL* for more information on setting dialects.

SQL uses the identifier character set as the character set for database object names. However, because SQL interprets the names of some database objects as file names or path names, you must use only ASCII alphanumeric characters for the names of the following database objects:

- Database file name
- Database path name
- Snapshot files
- Storage areas
- Journal files
- Alias names

If you do not use delimited identifiers, SQL considers uppercase and lowercase letters in database object names (other than file names) to be the same because it converts lowercase letters to uppercase. That is, EMPLOYEE_ID, employee_id, and Employee_ID are equivalent because SQL converts them to EMPLOYEE_ID. SQL does not perform conversions on character sets that do not use the concept of uppercase and lowercase characters.

Note

If you use an SQL keyword as a user-supplied name, delimit the name with double quotation marks to differentiate the name from a keyword. Not doing so can cause unexpected results.

Unlike some products, SQL does not convert a hyphen (which is interpreted as a minus sign) in user-supplied names to an underscore. Instead, it considers hyphens and underscores in such names to be distinct characters. This means you cannot use hyphens in user-supplied names. For instance, you cannot type EMPLOYEE-ID instead of the column name EMPLOYEE_ID.

Host language parameters in embedded SQL statements are a special case in which the SQL precompiler follows language-specific rules for user-supplied names. The precompiler follows the convention of the host language in distinguishing uppercase from lowercase letters, hyphens from underscores, and valid from invalid characters.

Table 2–5 gives brief definitions of user-supplied names referred to in syntax diagrams. Subsequent sections discuss many of these names in more detail.

Table 2–5 Summary of User-Supplied Names Used in SQL

User-Supplied Name	Description
area-name	A name that designates storage area and snapshot files that are associated with particular tables in a multifile database. You must use ASCII alphanumeric characters for the area name.
alias	A name for a particular attachment to a database. Using aliases, programs, or interactive SQL statements allow reference to more than one database in an environment. Aliases can, and sometimes must, qualify database definition names to distinguish them from another database's definitions. An alias name is restricted to a length of 31 characters. The name must begin with an alphabetic character and can contain numeric characters, the dollar sign ($), and the underscore (_) characters.
auth-id	A name used for identifying schemas in a multischema database and for checking privileges.
catalog-name	A name for a database object that contains one or more schemas. Databases that do not use the multischema option do not include any catalogs.
collation-name	The name by which the collating sequence named in the ncs-name argument of the CREATE COLLATING SEQUENCE statement will be known to the schema. The collation-name and ncs-name arguments can be the same.
column-name	A name that designates a column in a view or table definition. A column name can be qualified by a table name, view name, correlation name, or alias.
connection-name	A name that designates a connection. A connection specifies an association between the set of cursors, intermediate result tables, and procedures in all modules of an application and the database environment currently attached. When you execute a procedure, it executes in the context of a connection.

(continued on next page)

Table 2–5 (Cont.) Summary of User-Supplied Names Used in SQL

User-Supplied Name	Description
constraint-name[1]	A name that designates a constraint. A constraint specifies a condition that restricts the values stored in tables. When you insert and update column values, SQL checks the values against the conditions specified by the constraint. The insert or update statement fails if a value violates the constraint.
correlation-name	A temporary name that identifies a result table to SQL. A result table is a temporary set of rows and columns created by an SQL statement for a data manipulation operation. Correlation names qualify column names and distinguish between columns of different result tables, even if the columns have the same name.
currency-char	Specifies the currency indicator to be displayed in output.
cursor-name	A name that designates a cursor. A cursor identifies rows of a result table for processing by a program.
date-number	Specifies the input and 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 the OpenVMS run-time library documentation.
digit-sep-char	Changes the output that displays the digit separator to the specified character. The digit separator is the symbol that separates groups of 3 digits in values greater than 999. For example, the comma is the digit separator in the number 1,000.
domain-name[1]	A name that designates a domain. A domain definition restricts the set of values that a table column can have by associating a data type with a domain name, and allows optional formatting and collating clauses. Column definitions in tables and parameter declarations in SQL module language procedures can name a domain instead of specifying a data type.
external-routine-name[1]	A name that you assign to an external function or external procedure, which resides as a schema object in Oracle Rdb.
function-name[1]	A name that designates a stored function within a stored module. A stored function can only contain IN parameter declarations. When you use a value expression to call a stored function, you identify the function by its stored function name.

[1]You can qualify this object with an alias. In a multischema database, you can qualify this object with an alias, catalog name, and a schema name. To qualify an object, you must precede it with the qualifier and a period (.).

(continued on next page)

Table 2–5 (Cont.) Summary of User-Supplied Names Used in SQL

User-Supplied Name	Description
file-spec	A full or partial file specification that designates the source of data definitions and the location of database files. You must use ASCII alphanumeric characters for the file specification name. You can use a logical name in place of the file specification.
index-name[1]	A name that designates an index.
language-name	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. Valid entries for the language-name argument are the names of the collating sequences used by the National Character Set (NCS) utility.
library-name	The name of an NCS library other than the default. The default (ASCII) NCS library is SYS$LIBRARY:NCS$LIBRARY.
map-name	A name that designates a storage map that controls which rows and columns of a table are stored in which storage areas in a multifile database.

[1]You can qualify this object with an alias. In a multischema database, you can qualify this object with an alias, catalog name, and a schema name. To qualify an object, you must precede it with the qualifier and a period (.).

(continued on next page)

Table 2–5 (Cont.) Summary of User-Supplied Names Used in SQL

<table>
<tr><th>User-Supplied Name</th><th>Description</th></tr>
<tr><td>module-name</td><td>The name of the module.
For nonstored modules

A name that you assign to a nonstored module. Nonstored modules can contain simple or compound statement procedures that are called by host language programs. Unlike stored modules, nonstored modules reside outside an Oracle Rdb database in an SQL module file. If you omit the module-name, SQL uses SQL_MODULE by default.
For stored modules

A name that you assign to a module that resides as a schema object in an Oracle Rdb database. Stored modules can contain compound statement procedures only, which a host language program calls from a simple statement procedure using the CALL statement. When you define a stored module with the CREATE MODULE statement, you also define its functions or procedures, which are called stored functions or stored procedures. You must specify a module name; otherwise, SQL returns an exception.</td></tr>
<tr><td>ncs-name</td><td>The name of a collating sequence in the default NCS library, SYS$LIBRARY:NCS$LIBRARY, or in the NCS library specified by the argument library-name. The collating sequence can be either one of the predefined NCS collating sequences or one that you defined yourself using the NCS collating sequences.</td></tr>
<tr><td>parameter</td><td>A variable declared in a host language program that is associated with an SQL statement, including:
host language variables in precompiled programs
formal parameters in SQL module procedures
parameter markers in dynamic SQL</td></tr>
<tr><td>path-name</td><td>A full or relative data dictionary path name that specifies the source of schema definitions. You must use ASCII alphanumeric characters for the path name.</td></tr>
</table>

(continued on next page)

Table 2–5 (Cont.) Summary of User-Supplied Names Used in SQL

<table>
<tr><th>User-Supplied Name</th><th>Description</th></tr>
<tr><td>procedure-name</td><td>A name that designates a stored or nonstored procedure within a stored or nonstored module:
Stored procedures
Can contain zero or more parameter declarations and a compound statement. When you use the CALL statement to call a stored procedure, you identify the procedure by its stored procedure name.
Nonstored procedures
Can contain one or more parameter declarations and a simple or compound statement. Nonstored procedure names are used in host language calls to the SQL module.</td></tr>
<tr><td>profile-name[1]</td><td>A name that designates a profile.</td></tr>
<tr><td>radix-char</td><td>Changes the output that displays 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.</td></tr>
<tr><td>role-name[1]</td><td>A name that designates a role.</td></tr>
<tr><td>schema-name</td><td>A name that designates a schema. A schema specifies a group of data definitions within a database. In a multischema database, one or more schemas are grouped together within catalogs.</td></tr>
<tr><td>sequence-name[1]</td><td>A name that designates a sequence.</td></tr>
<tr><td>statement-name</td><td>A name that designates a prepared SQL statement. A prepared statement is one generated dynamically during the execution of a program.</td></tr>
<tr><td>synonym-name[1]</td><td>A name that designates a synonym.</td></tr>
<tr><td>table-name[1]</td><td>A name that designates a table in which data is stored. A table name can qualify a column name.</td></tr>
<tr><td>time-number</td><td>Specifies the input and 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 the OpenVMS run-time library documentation.</td></tr>
</table>

[1]You can qualify this object with an alias. In a multischema database, you can qualify this object with an alias, catalog name, and a schema name. To qualify an object, you must precede it with the qualifier and a period (.).

(continued on next page)

Table 2–5 (Cont.) Summary of User-Supplied Names Used in SQL

User-Supplied Name	Description
trigger-name[1]	A name that designates a trigger definition. A trigger definition causes one or more actions to occur when a particular type of update operation is performed on the table. A trigger name must be unique within a schema.
user-name[1]	A name that designates a user.
view-name[1]	A name that designates a view. A view is a table whose data is not physically stored but refers to rows, columns, or both, stored in other tables. A view name can qualify a column name.

[1]You can qualify this object with an alias. In a multischema database, you can qualify this object with an alias, catalog name, and a schema name. To qualify an object, you must precede it with the qualifier and a period (.).

2.2.1 Aliases

An **alias** is a name for a particular attachment to a database. Explicitly specifying an alias lets your program or interactive SQL statements refer to more than one database.

Once you specified the alias, you must use it when referring to the database in subsequent SQL statements (unless those statements are within a CREATE DATABASE statement). You must use an alias when you declare more than one database so that SQL knows the database to which your statements refer. When you issue an ATTACH, CONNECT, CREATE DATABASE, CREATE DOMAIN, CREATE TABLE, DECLARE ALIAS, GRANT, GRANT (ANSI-style), IMPORT, REVOKE, or SET TRANSACTION statement, you can specify an alias in addition to a file specification or a repository path name.

SQL allows you to specify an alias that declares the database as the default database. Specifying a **default database** means that subsequent statements that refer to the default database during the database attachment do not need to use an alias.

In the SQL module language, the alias specified in the module header designates the default database. In precompiled SQL programs and in interactive SQL, the special alias RDB$DBHANDLE designates the default database if you are not using multischema naming. To use an alias with a multischema database, you must use the QUOTING RULES SQL99 clause in the module header, and you must use the delimited identifiers described in Section 2.2.11. In all environments, omitting an explicit alias is the same as specifying the alias that designates the default database. If you do not declare an alias, SQL uses the database file specified by the logical name SQL$DATABASE as the default database for module compilation.

If you declare an alias that designates a database other than the default database, you must use that alias to qualify names of any database objects (tables, views, indexes, domains, storage maps, storage areas) to which you refer in SQL statements. If you omit the alias, SQL assumes the database object is part of the default database. If there is no default database and you omit the alias, SQL generates an error. See Section 2.2.19 for an example of qualifying a table name with an alias.

The following example shows how you can specify aliases in ATTACH statements. One of the databases is empty and will be used to make temporary copies of tables in the personnel database. Use the SHOW DATABASE statement to see the database settings.

```
SQL> -- Use the alias empty for the empty database.
SQL> --
SQL> ATTACH 'ALIAS empty PATHNAME temp';
SQL> --
SQL> -- Use the alias pers for personnel.
SQL> --
SQL> ATTACH 'ALIAS pers PATHNAME personnel';
SQL> --
SQL> -- You must use the alias to qualify table names after you declare
SQL> -- an alias.  Omitting the alias generates an error.
SQL> --
SQL> SELECT * FROM EMPLOYEES;
%SQL-F-NODEFDB, There is no default database
SQL> SELECT * FROM PERS.EMPLOYEES;
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
   .
   .
   .
```

2.2.2 Authorization Identifiers

SQL uses an **authorization identifier** in a stored or nonstored module to convey to Oracle Rdb the concept of a user. These modules can be either definer's rights or invoker's rights.

2.2.2.1 Authorization Identifiers and Stored Modules

A stored module resides in the database as an object. You can store modules and their procedures and functions with the CREATE MODULE statement.

The authorization identifier, specified by using the AUTHORIZATION clause, enables Oracle Rdb to identify the user under whom the module executes.

When you specify an authorization identifier in the definition of a stored module, that stored module is called a **definer's rights module**. This type of module enables any user who has EXECUTE privilege on the module to execute any of the module's routines without privileges on any of the underlying schema objects that the routine references. The routines execute under the rights identifier of the module definer, not the rights identifier of the person executing the routine. This ability to allow users access to schema objects through a call to a stored routine without having direct access to those schema objects is a key benefit of stored modules.

In contrast, when you omit the AUTHORIZATION clause in the definition of a stored module, that stored module is called an **invoker's rights module**. In this type of module, users who have EXECUTE privilege on a particular module must also have privileges to all the underlying schema objects associated with any of the routines in this module that they want to execute.

The following examples relate to stored modules and procedures. Authorization and CURRENT_USER are handled the same for both types of stored routines.

Consider the following stored module definition, Module M1 with Procedure P1 and Authorization Brown. For example:

```
CREATE MODULE M1
    LANGUAGE SQL
    AUTHORIZATION BROWN

    PROCEDURE P1 ();
    BEGIN
    TRACE CURRENT_USER;
    CALL P2 ();
    END;
END MODULE;
```

As you can see in the preceding example, P1 calls another stored procedure, P2. Procedure P2 is defined in Module M2 as the following example shows:

```
CREATE MODULE M2
    LANGUAGE SQL
    -- no authorization
    PROCEDURE P2 ();
    BEGIN
    TRACE CURRENT_USER;
    CALL P3 ();
    END;
END MODULE;
```

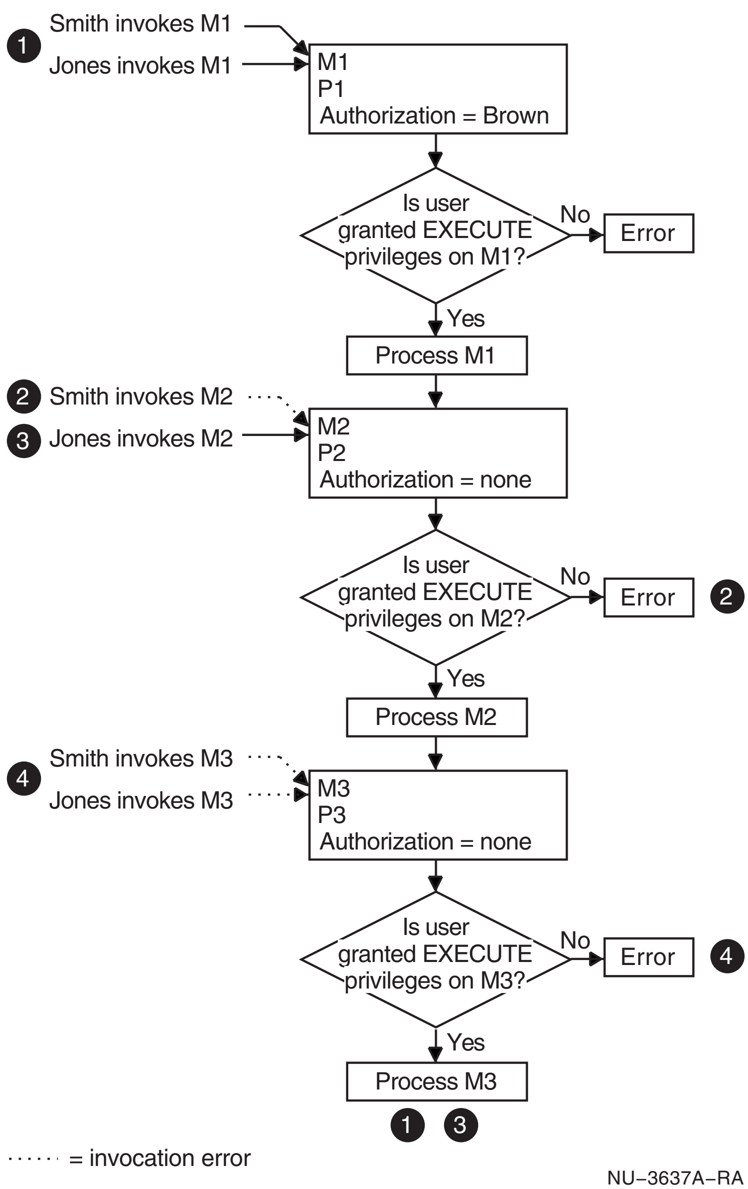
Procedure P2 calls another procedure, P3, from Module M3, which is shown in the following example:

```
CREATE MODULE M3
    LANGUAGE SQL
    -- no authorization
    PROCEDURE P3 ();
    BEGIN
    TRACE CURRENT_USER;
    .
    .
    .
    END;
END MODULE;
```

In each procedure you can trace the CURRENT_USER.

Figure 2–1 is a graphic representation of what happens when users invoke these stored procedures.

Figure 2–1 Authorization Identifiers and Stored Modules



Assume the following:

- Smith is granted the EXECUTE privilege on Module M1; but not on M2 or on M3.
- Brown is granted the EXECUTE privilege on Modules M1, M2, and M3.
- Jones is granted the EXECUTE privilege on Modules M1 and M2.

When P1 is executed, CURRENT_USER always returns Brown as defined by the AUTHORIZATION clause in Module M1. When P2 or P3 are executed, the CURRENT_USER is either:

- Inherited from the calling routines AUTHORIZATION clause, or
- The CURRENT_USER of the calling routine if no authorization was specified

When there is no AUTHORIZATION clause for the first calling routine, then CURRENT_USER is inherited from the SESSION_USER.

The following list explains the numbered callouts in Figure 2–1.

1 When Smith and Jones invoke P1, the routine executes under the authorization of Brown. P1 then calls P2. Brown is granted access to Module M2 and can, therefore, execute P2 giving Smith and Jones implicit access to P2. When referenced, CURRENT_USER in P2 inherits the current user of the calling routine, which is Brown.

2 When Smith tries to execute Procedure P2 directly, an error is returned because Smith does not have EXECUTE privilege on Module M2.

3 When Jones executes P2 directly, CURRENT_USER is displayed as Jones (inherited from SESSION_USER because Module M2 was defined without an authorization identifier). P2 can have a different CURRENT_USER depending on how it is invoked.

  From a security point of view, when Jones executes P2 directly, Jones must have EXECUTE privilege on Module M2. However, when Jones executes P2 using a call from P1, then Brown must have EXECUTE privilege on Module M2, and Jones must have EXECUTE privilege on Module M1.

4 When Smith and Jones try to execute Procedure P3 directly, an error is returned because they do not have EXECUTE privileges on Module M3.

2.2.2.2 Authorization Identifiers and Nonstored Modules

A nonstored module resides outside the database in an SQL module file.

The AUTHORIZATION clause specifies the authorization identifier for the module. If you omit the authorization identifier, SQL selects the user name of the user compiling the module as the default authorization. Thus, if you use the RIGHTS clause, SQL compares the user name of the person who executes a module with the authorization identifier with which the module was compiled and prevents any user other than the one who compiled that module from invoking that module. When you use the RIGHTS clause, SQL bases privilege checking on the default authorization identifier in compliance with the ANSI/ISO standard.

2.2.3 Catalog Names

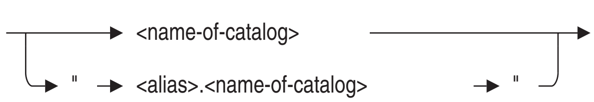
If you include the MULTISCHEMA IS ON clause in your CREATE DATABASE statement, you can store your metadata in multiple schemas. A database with multiple schemas must organize them within catalogs. A **catalog** is a group of schemas within one database.

You name catalogs in CREATE CATALOG or CREATE DATABASE statements. You can also use catalog names to qualify the names of other database elements such as schemas, tables, and views.

Note

In syntax diagrams, the column-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the catalog in the CREATE statement. That is, in syntax diagrams, the catalog-name is always defined as:

catalog-name =



In each multischema database, SQL creates a catalog named RDB$CATALOG. SQL stores all schemas in RDB$CATALOG by default. A multischema database must contain at least one catalog, although you can create more than one catalog for each database. To store a schema in a catalog other than RDB$CATALOG, qualify the schema name with the other catalog's name in the CREATE SCHEMA statement, or use the SET CATALOG statement to change the default catalog before issuing a CREATE SCHEMA statement.

In the following example, SQL puts the new schema PACIFIC_NORTHWEST into the default catalog, RDB$CATALOG. To create a schema in the EAST_COAST catalog, you must use the catalog name EAST_COAST to qualify the schema NEW_ENGLAND. If you change the default catalog to EAST_COAST, you must qualify names of schemas in other catalogs, such as RDB$CATALOG.

```
SQL> ATTACH 'FILENAME corporate_data';
SQL> CREATE SCHEMA PACIFIC_NORTHWEST;
SQL> CREATE CATALOG EAST_COAST;
SQL> CREATE SCHEMA EAST_COAST.NEW_ENGLAND;
SQL> SHOW SCHEMAS;
Schemas in database with filename corporate_data
    ADMINISTRATION.ACCOUNTING
    ADMINISTRATION.PERSONNEL
    ADMINISTRATION.RECRUITING
    EAST_COAST.NEW_ENGLAND
    PACIFIC_NORTHWEST
    RDB$SCHEMA
SQL> SET CATALOG 'EAST_COAST';
SQL> SHOW SCHEMAS;
Schemas in database with filename corporate_data
    ADMINISTRATION.ACCOUNTING
    ADMINISTRATION.PERSONNEL
    ADMINISTRATION.RECRUITING
    NEW_ENGLAND
    RDB$CATALOG.PACIFIC_NORTHWEST
    RDB$CATALOG.RDB$SCHEMA
```

Within a 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 specifying an object in a multischema database, SQL uses the name of the current default catalog.

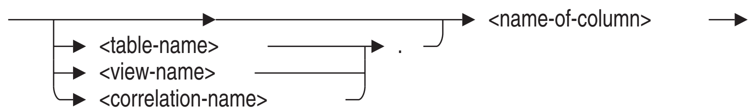
2.2.4 Column Names

You name columns in CREATE TABLE and ALTER TABLE statements. In other SQL statements, the names you give to columns in CREATE and ALTER statements can be qualified by table names, view names, or correlation names.

Note

In syntax diagrams, the column-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the column in the CREATE TABLE or ALTER TABLE statement. That is, in syntax diagrams, column-name is always defined as:

column-name =



The only time you must qualify column names is when they are ambiguous. Joining a table with itself (see Section 2.2.4.1 for an example) and joining two tables with common column names (see the following example) are two cases that require qualified column names. Also, if you have a parameter without a colon with the same name as a column, you need to qualify references to that column.

However, you always have the option of qualifying column names. In complex statements, such qualifiers often make the statements more readable. (You should always qualify column names in module language and precompiled programs. Otherwise, you will need to revise the program and qualify the column names if changes make the unqualified column ambiguous.)

There are two ways to qualify column names:

- With the name of the table or view to which the column belongs.
- With an arbitrary correlation name you specify. You must use correlation names instead of table names or view names when you join a table with itself. Once you specify a correlation name for a table, you can no longer use the table name or view name to qualify column names.

The column-name qualifier (whether a table name or a view name) can itself be qualified by an alias.

The remainder of this section gives examples of qualifying column names with table names or view names. See Section 2.2.4.1 for examples of using correlation names.

The following example illustrates optional qualification of column names. The query does not join tables because it retrieves column values from the EMPLOYEES table only. Instead, it nests a select expression in the predicate of another statement to list the employees who work in the marketing department. The query does not require qualifiers on the column names but uses them to clearly distinguish to which table the EMPLOYEE_ID column refers.

```
SQL> SELECT    EMPLOYEE_ID, FIRST_NAME, LAST_NAME
cont> FROM      EMPLOYEES
cont> WHERE     EMPLOYEES.EMPLOYEE_ID IN
cont>           (SELECT JOB_HISTORY.EMPLOYEE_ID
cont>            FROM   JOB_HISTORY
cont>            WHERE  JOB_END IS NULL
cont>            AND    DEPARTMENT_CODE = 'MKTG');
 EMPLOYEE_ID   FIRST_NAME   LAST_NAME
 00197         Chris        Danzig
 00218         Lawrence     Hall
 00354         Paul         Belliveau
3 rows selected
```

The following example retrieves the same information as the previous example but illustrates a case when you must qualify a column name, EMPLOYEE_ID, because it is ambiguous. The SELECT statement joins the EMPLOYEES and JOB_HISTORY tables to list the employees who work in the marketing department. Because both EMPLOYEES and JOB_HISTORY have a column called EMPLOYEE_ID, that column must be qualified.

```
SQL> SELECT EMPLOYEES.EMPLOYEE_ID,
cont>   FIRST_NAME, LAST_NAME
cont>      FROM EMPLOYEES, JOB_HISTORY
cont>      WHERE JOB_END IS NULL
cont>      AND
cont>         DEPARTMENT_CODE = 'MKTG'
cont>      AND
cont>         JOB_HISTORY.EMPLOYEE_ID = EMPLOYEES.EMPLOYEE_ID;
 EMPLOYEES.EMPLOYEE_ID   EMPLOYEES.FIRST_NAME   EMPLOYEES.LAST_NAME
 00197                   Chris                  Danzig
 00218                   Lawrence               Hall
 00354                   Paul                   Belliveau
3 rows selected
```

2.2.4.1 Correlation Names

In addition to qualifying column names with table names or view names, you can qualify column names with correlation names. **Correlation names** are analogous to aliases, but they refer to tables instead of databases. Just as aliases provide temporary names for databases to qualify ambiguous table names, correlation names give temporary names to tables to qualify ambiguous column names.

Specify a correlation name after a table name within the FROM clause of a select expression or DELETE statement, or in an UPDATE statement. Use any valid name that has not already been used in the FROM clause either as a correlation name or as a table name without a correlation name.

You must use correlation names to qualify column names in statements that join a table with itself. As with table names and view names, however, you can always specify a correlation name for clarity (Section 2.2.4.2 shows an example of this within an outer reference).

The following example requires the use of a correlation name. It joins the JOBS table with itself to find any wage class 2 jobs whose maximum salary overlaps the minimum salary of wage class 4 jobs.

The statement specifies the correlation names STAFF and MGR in the FROM clause. Those correlation names are the only way to distinguish between column names in the result table that joins JOBS with itself.

```
SQL> SELECT    STAFF.JOB_CODE,
cont>           STAFF.MAXIMUM_SALARY,
cont>           MGR.JOB_CODE,
cont>           MGR.MINIMUM_SALARY
cont>   FROM    JOBS AS STAFF,
cont>           JOBS AS MGR
cont>   WHERE   MGR.WAGE_CLASS = '4'
cont>           AND
cont>           STAFF.WAGE_CLASS = '2'
cont>           AND
cont>           STAFF.MAXIMUM_SALARY > MGR.MINIMUM_SALARY;
 STAFF.JOB_CODE   STAFF.MAXIMUM_SALARY   MGR.JOB_CODE   MGR.MINIMUM_SALARY
 CLRK                         20000.00   APGM                     15000.00
1 row selected
```

The example shows that the maximum salary for a clerk is greater than the minimum salary for an associate programmer. Those two are the only jobs where the maximum pay for a wage class 2 job exceeds the minimum for a wage class 4 job.

In the absence of an explicit correlation name, SQL considers table names or view names as default correlation names, even if you do not use the table names or view names to explicitly qualify column names in the select list. Because of this, SQL generates an error if you name the same table twice in the FROM clause without specifying a correlation name.

```
SELECT JOB_CODE, MINIMUM_SALARY FROM JOBS, JOBS;
%SQL-F-CONVARDEF, Column qualifier JOBS is already defined
```

In this example, because no correlation name was specified, SQL by default considers JOBS as the qualifier for the first occurrence of the JOBS table. When SQL encounters the second occurrence of JOBS, also without a correlation name, it generates an error because it uses the second JOBS as a second, ambiguous default correlation name. To prevent the error, specify a correlation name for either occurrence of JOBS in the FROM clause, and then qualify column names in the select list.

Once you specify a correlation name for a table, you can no longer use the table name to qualify column names. The following example specifies E as a correlation name for the EMPLOYEES table, which means EMPLOYEES cannot be used as a qualifier for the EMPLOYEE_ID column name:

```
SELECT * FROM EMPLOYEES E WHERE EMPLOYEES.EMPLOYEE_ID = '00169';
%SQL-F-CONVARUND, Column qualifier EMPLOYEES is not defined
```

2.2.4.2 Outer References

You may have to qualify column names in an outer reference. An **outer reference** is a reference within a subquery to a table specified in an outer query that contains the subquery. An outer reference is also called a correlated reference.

For example, the previous example that retrieved the names of employees who worked in the marketing department can be reformulated to use an outer reference.

```
SQL> SELECT    FIRST_NAME,                             --
cont>           LAST_NAME                              --
cont>   FROM    EMPLOYEES                              --
cont>   WHERE   'MKTG' IN                              --
cont>           (SELECT DEPARTMENT_CODE        --       -- Outer
cont>            FROM   JOB_HISTORY            --       -- Query
cont>            WHERE  JOB_END IS NULL        -- Sub-  --
cont>                   AND                    -- query --
cont>                   EMPLOYEE_ID =
cont>                   EMPLOYEES.EMPLOYEE_ID)  --       --
cont> --                ---------------------
cont> --                  outer reference
cont> ;
 FIRST_NAME   LAST_NAME
 Chris        Danzig
 Lawrence     Hall
 Paul         Belliveau
3 rows selected
```

If the outer reference to EMPLOYEE_ID in this example were not qualified by the table name EMPLOYEES, it would refer to the EMPLOYEE_ID column in the subquery, not the outer query. The predicate EMPLOYEE_ID = EMPLOYEE_ID is true for all values of EMPLOYEE_ID that are not null, so the statement would not generate an error, but would give unexpected results. Instead of the three marketing employees, it would select all rows of the EMPLOYEES table with values in the EMPLOYEE_ID column that were not null.

Although the outer reference is contained within a subquery, it receives its value from an outer query. Because of this, the subquery must be evaluated once for each value that the outer reference receives from the outer query. It is this characteristic that defines an outer reference.

In the previous example, the outer reference in the last line of the statement EMPLOYEES.EMPLOYEE_ID gets a different value for each row of the table EMPLOYEES. SQL evaluates the subquery containing EMPLOYEES.EMPLOYEE_ID once for every value of EMPLOYEE_ID in the table EMPLOYEES.

To make the correlation between the reference in the subquery and the table in the outer query clearer, you can specify correlation names, such as MAIN_QUERY and SUBQUERY in the following example:

```
SQL> SELECT MAIN_QUERY.FIRST_NAME,
cont> MAIN_QUERY.LAST_NAME
cont>   FROM EMPLOYEES MAIN_QUERY
cont>   WHERE 'MKTG' IN
cont>      (SELECT SUBQUERY.DEPARTMENT_CODE
cont>         FROM JOB_HISTORY SUBQUERY
cont>         WHERE SUBQUERY.JOB_END IS NULL
cont>         AND
cont>         SUBQUERY.EMPLOYEE_ID = MAIN_QUERY.EMPLOYEE_ID);
 FIRST_NAME   LAST_NAME
 Chris        Danzig
 Lawrence     Hall
 Paul         Belliveau
3 rows selected
```

2.2.5 Connection Names

When your application attaches to one or more databases, SQL associates the databases with a set of aliases (database handles). In CONNECT, DISCONNECT, or SET CONNECT statements, you refer to this association as the **connection name**. You can specify the connection name as a parameter marker from dynamic SQL, a host language variable from a precompiled SQL program, a parameter from an SQL module language module, or a string literal.

The set of databases that you can attach or detach as one unit is called the **database environment**. Within an application, all of the databases declared in all the modules form the default database environment for that application at run time. For more information about connections, see the CONNECT Statement.

2.2.6 Constraint Names

A **constraint** defines a condition that restricts the values that can be stored in a table. When you insert and update column values, the constraint checks the values against the conditions specified by the constraint. If a value violates the constraint, SQL generates an error message and the statement fails (either when the INSERT, UPDATE, or DELETE statement executes, or when the next COMMIT statement executes depending on when SQL evaluates the constraint).

You specify constraints in CREATE and ALTER TABLE statements. Optionally, you supply a name for the constraints following the CONSTRAINT keyword.

2.2.7 Cursor Names

Cursors provide access to individual rows of a result table. A **result table** is a temporary collection of columns and rows from one or more tables or views. For cursors, the result table is specified by the select expression in the DECLARE CURSOR statement.

Unlike other result tables, the result table for a cursor can exist throughout execution of more than one statement. Host language programs require cursors because programs must perform operations one row at a time, and therefore can execute statements more than once to process an entire result table.

You name the result table for a cursor in the DECLARE CURSOR statement and refer to that name in OPEN, CLOSE, FETCH, UPDATE, and DELETE statements. You cannot qualify cursor names.

2.2.8 Database Names

A **database** consists of physical data storage characteristics, such as a root file and storage area specifications; metadata definitions, such as tables and domains; and user data.

By default, a database contains a single schema and no catalogs. If you specify the multischema attribute when creating your database, you can group the data definitions within one or more schemas within one or more catalogs. See the CREATE DATABASE Statement for information on how to create a multischema database.

When you create a database, you name it by specifying a file name and an optional repository path name in the CREATE DATABASE statement. You can supply a complete file specification, a partial file specification, or use system-supplied default values. You must use ASCII alphanumeric characters for the database name.

To perform operations on a database, the database name is referenced through an attachment to that database called an **alias**. When you first refer to a database in SQL, you must indicate the source of data definitions for the database and the location of database files by declaring an alias. You can declare an alias using one of three statements:

- ATTACH
- CONNECT
- DECLARE ALIAS

Choose a statement based on the interface that you are using (interactive SQL, SQL module language, or precompiled SQL) and your purpose (declaring a new alias or overriding the association between an alias and a database name). For details, see the statements in Chapter 6 and Chapter 7. More information about aliases appears in Section 2.2.1.

There are two ways to identify the source of data definitions:

- With a file specification
- With a repository path name (if the repository is installed on the system)

The following sections describe these methods in more detail.

2.2.8.1 Oracle Rdb Attach Specifications

When you first create a database, you give file specifications for the files that contain all database definitions (metadata) and user data stored in the database. You must use ASCII alphanumeric characters for the file specification name.

You can also use a file specification whenever you refer to a database in the CONNECT and DECLARE ALIAS statements, although Oracle Rdb recommends that you always use a repository path name when the repository is installed. See the Usage Notes in the DECLARE ALIAS Statement and the CONNECT Statement for more information.

A full file specification includes:

- Network node name
- Device name
- Directory name or list
- File name
- File extension
- File version number

For example, on OpenVMS a full file specification is:

```
SPEEDY::DISK_DEPT3:[LICENSES]APPLICANTS.RDB;18
```

If it can, the system supplies default values for omitted fields in the file specification.

When you use a CREATE DATABASE, ALTER DATABASE, DROP DATABASE, ATTACH, or DECLARE ALIAS statement, you should not include a file extension or version number in the file specification. The file specification is used to create other files with different file extensions. For example, on OpenVMS, when you create a single-file database, Oracle Rdb creates two files: one with an .rdb file extension and one with an .snp file extension. Specifying an extension or version can cause mismatches between the two files.

You can also use logical names instead of full or partial file specifications in an ATTACH or a DECLARE ALIAS statement.

If you are using a remote database (that is, a database on another node in a network), you must be sure to include the node name in the file specification. Because access to a remote database requires use of another computer system, your process or program must somehow log in to that system and authenticate the user.

To access databases on remote nodes, you can explicitly provide user name and password information in SQL statements that attach to the database and in configuration parameters. In addition, you can pass the information to an SQL module language or precompiled SQL program by using a parameter and a new command line qualifier.

When you use Oracle Rdb for OpenVMS to attach to a database in the same cluster, you do not have to explicitly specify the user name and password. Oracle Rdb implicitly authenticates the user whenever the user attaches to a database.

However, when you use Oracle Rdb for OpenVMS to attach to a database on a remote node, even if that node is an OpenVMS node, you must use one of the methods provided by Oracle Rdb to access the database.

You can use one of the following methods to attach to a database on a a remote OpenVMS node:

- You can explicitly provide the user name and password in the USER and USING clauses of the ATTACH statement.

To attach to the mf_personnel database on a remote node, you can use the USER and USING clauses in the ATTACH statement, as the following example shows:

```
SQL> ATTACH 'FILENAME REMNOD::DISK1:[DIR]MF_PERSONNEL
cont>        USER ''heleng'' USING ''MYpassword''';
SQL>
```

You must enclose the user name and password in single quotation marks, but because the literal in this example is within the quoted attach-string, you must surround the user name and password with two sets of single quotation marks.

- Explicitly provide the user name and password in the configuration file RDB$CLIENT_DEFAULTS.DAT. The following example shows how to include the information in the configuration file:

```
! User name to be used for authentication
SQL_USERNAME   HELENG

! Password to be used for authentication
SQL_PASSWORD  MYPASSWORD
```

- You can also use one of the following methods to attach to a database on a remote OpenVMS node:
  - Use a proxy account on the remote system. Grant database privilege to the RDB$REMOTE account created by the Oracle Rdb installation. For more information, see the *Oracle Rdb Guide to SQL Programming*.
  - Use a proxy account. Such an account need not have the same privileges as the local account and is the recommended method of remote access. This approach eliminates the need to include a user name and password in an ATTACH statement in a command file or in a DECLARE ALIAS statement in a host language program. For more information about proxy accounts, see the *Oracle Rdb Guide to SQL Programming*.
  - Embed a user name and password in the file specification. The following example shows the ATTACH statement for access to the remote system REMNOD:

```
SQL> ATTACH 'FILENAME REMNOD"JULIA OPEN_UP"::APPLICANTS.RDB';
```

Here, REMNOD is the remote node name, JULIA is the user name for the account in which the database is defined, and OPEN_UP is the password for that account. No disk or directory specification is required if the database files are in Julia's login directory. DECnet software runs the login process for the user named JULIA and uses Julia's disk and login directory automatically.

If you do not specify the USER and USING clause in SQL statements, Oracle Rdb uses the information in the configuration file. If the SQL_USERNAME and SQL_PASSWORD parameters are not specified in the configuration file, Oracle Rdb checks for the existence of proxy accounts.

2.2.8.2 Repository Path Names

Unless you use the PATHNAME argument in the CREATE DATABASE statement, SQL does not use the repository to store data definitions.

If you specify the PATHNAME argument when you first create a database, SQL creates a path name that contains copies of data definitions for the database.

Because SQL treats a path name like a string literal, you must enclose a path name in single quotation marks. You must use ASCII alphanumeric characters for the repository path name.

When you issue an ATTACH or a DECLARE ALIAS statement, you can either specify the repository path name for that database (which in turn points to the physical database files) or directly name the physical database file specification.

If you do not use the PATHNAME argument in the CREATE DATABASE statement, you cannot specify a path name in ATTACH or DECLARE ALIAS statements for that database unless you first issue an INTEGRATE statement. Oracle Rdb recommends that you always use a repository path name in CREATE DATABASE, ATTACH, and DECLARE ALIAS statements, and that you use the DICTIONARY IS REQUIRED clause to ensure that the two copies are the same.

A repository path name can be a:

- Full path name, such as CDD$TOP.ELLINGSWORTH.SQL.PERSONNEL
- Relative path name

  A relative path name consists of the portion of the full path name that follows the current default repository node. For example, assume that you used the SET DICTIONARY command to set the current repository directory to CDD$TOP.ELLINGSWORTH.SQL. Now you can use the relative path name PERSONNEL in place of the full path name CDD$TOP.ELLINGSWORTH.SQL.PERSONNEL. By default, SQL sets the current repository node to the path name defined by the CDD$DEFAULT logical name. See the SET Statement for the description of the SET DICTIONARY statement. See also *Using Oracle CDD/Repository on OpenVMS Systems* for more detail on repository path names.

- Logical name for a full or relative path name

Some Oracle Rdb features are not fully supported by all versions of the repository. If you attach by path name and attempt to create, modify, or delete objects not fully supported by the repository, you may receive an error or informational message. See the *Oracle Rdb Release Notes* for information about compatibility of Oracle Rdb features with the different versions of the repository.

2.2.9 Domain Names

A **domain** is the set of values that a table column can have.

A domain definition restricts the set of values that a table column can have by associating a data type with a domain name, and allows optional formatting and collating clauses. The CREATE and ALTER TABLE statements refer to domain names in column definitions. The domain name must be unique among domain names in the schema.

You can use a domain when defining columns in multiple tables. Once you have defined a domain, use the CREATE or ALTER TABLE statement to define a column based on the domain definition.

You can qualify the domain name with the schema name (when the domain belongs to a multischema database) or with the alias.

In general, you should use domains when you create tables. Using domains:

- Ensures that similar columns in multiple tables comply to one standard. For example, if you define the columns using the domain ID_DOM, the data type for all these columns is CHAR(5).
- Allows you to change the data type for all columns defined using a domain by changing the domain itself. For example, if you want to change the data type for POSTAL_CODE_DOM from CHAR(5) to CHAR(10), you only need to alter the data type for POSTAL_CODE_DOM. You do not have to alter the data type for the column POSTAL_CODE in the tables COLLEGES and EMPLOYEES.

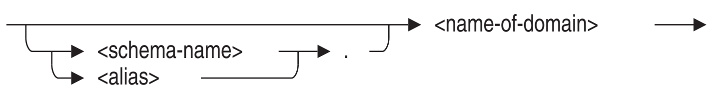
You might not want to use domains when you create tables if:

- You are creating intermediate result tables. It takes time to plan what the domains are in the database and to define them. Intermediate result tables might not warrant this effort.

Note

In syntax diagrams, the domain-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the domain in the CREATE DOMAIN statement.

domain-name =



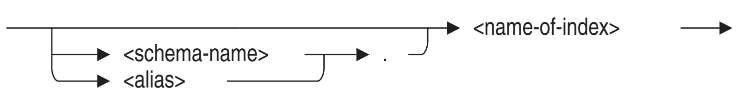
2.2.10 Index Names

You name indexes in the CREATE INDEX statement. In CREATE INDEX and other SQL statements, the names you give to indexes can be qualified by authorization identifiers.

Note

In syntax diagrams, the index-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the index in the CREATE INDEX statement.

index-name =



2.2.11 Names in Multischema Databases

If you specify the multischema attribute for your database, you can store data definitions in multiple schemas within that database. To specify the multischema attribute, use the MULTISCHEMA IS ON clause in a CREATE DATABASE or ALTER DATABASE statement. If you want SQL behavior compliant with the ANSI/ISO standard, you must specify the multischema attribute.

Databases that contain multiple schemas must organize the schemas within one or more catalogs. To refer to data definitions in a multischema database, qualify the names of data definitions with the schema and catalog names and, optionally, qualify with an alias.

When you use an alias to qualify the name of a catalog, schema, or object in a multischema database:

- Separate subordinate names from the alias and from each other with a period (.) after each name.
- Use double quotation marks (") to delimit the leftmost name pair.
- Use only uppercase characters in the leftmost name pair.

The leftmost name pair in a qualified name for a multischema object is a delimited identifier. In an object name, each qualifying name is considered one level, and names with more than three levels are not allowed. However, a delimited identifier is interpreted as a single level.

Any piece of a three-level name can have an alias embedded within double quotation marks, but you can only embed the alias in the leftmost level. For example, if you include the schema name but no catalog name (implying the default catalog), you can qualify the schema name with the alias using a delimited identifier.

By default, the Oracle Rdb implementation of SQL considers strings enclosed in double quotation marks to be string literals, but the ANSI/ISO SQL standard interprets strings enclosed by double quotation marks as delimited identifiers. To take advantage of the ANSI/ISO SQL standard, you must enable ANSI/ISO SQL quoting before you issue any statements that contain delimited identifiers. See Section 2.2 for information on how to enable ANSI/ISO SQL quoting.

Remember that the double-quoted leftmost pair in a multischema object name requires uppercase characters.

The following example shows a three-level name. CORPORATE is an alias for a database that contains the catalog MARKETING, the schema JONES, and the domain LAST_NAME.

```
SQL> SET QUOTING RULES 'SQL99';
SQL> SHOW DOMAIN "CORPORATE.MARKETING".JONES.LAST_NAME;
```

If the default catalog is set to MARKETING, user JONES can refer to the domain in the previous example using an object name qualified by the alias.

```
SQL> SET QUOTING RULES 'SQL99';
SQL> SHOW DOMAIN "CORPORATE.LAST_NAME";
```

Data definitions in single-schema and multischema databases follow different naming conventions. You can use the MULTISCHEMA IS OFF clause of the ATTACH or DECLARE ALIAS statement to disable multischema naming. Section 2.2.18 contrasts single-schema and multischema naming conventions. To specify RDB$SCHEMA or another schema name, you must attach to a multischema database with multischema naming enabled. Without multischema naming, you will only be able to refer to the entire database, using the alias associated with the database name.

If you do not specify the MULTISCHEMA IS clause, SQL enables multischema naming if the database was created with the multischema attribute and disables multischema naming if it was not.

The following example shows the error SQL generates if you try to create a schema in a database without the multischema attribute:

```
SQL> ATTACH 'FILENAME personnel';
SQL> CREATE SCHEMA PACIFIC_NORTHWEST;
%SQL-F-SCHCATMULTI, Schemas and catalogs may only be referenced
with multischema enabled
```

2.2.12 Nonstored Module, Procedure, and Parameter Names (Module Language Only)

The SQL module language provides a calling mechanism for host language programs to execute SQL statements contained in a separate file called an **SQL module file**. The module contains SQL statements that can be called from any host language, including those not supported by the SQL precompiler. The file contains module language elements, including the following user-supplied names:

- Module name

  You supply a module name after the MODULE keyword at the beginning of an SQL module. If you do not supply a module name, SQL names the module SQL_MODULE.

  Module names must be unique. The following error is returned if a nonstored module is invoked while a stored module with the same name is active:

  ```
  %RDB-E-EXT_ERR, Rdb extension error
  -RDMS-E-MODEXTS, there is another module named SALARY_ROUTINES in this
  database
  ```

- Procedure name

Every SQL module contains one or more procedures consisting of a procedure name, one or more actual parameter declarations, and a single executable SQL statement. You must supply a name for each procedure after the PROCEDURE keyword.

- Parameter name

  Actual parameters within a procedure in an SQL module specify a name to be used for the parameter by the SQL statement in the procedure. Some special-purpose procedure parameters are SQL keywords (SQLCODE, SQLCA, SQLDA, SQLSTATE), but you must give names to all other parameters in SQL modules.

See Chapter 3 for more information about the SQL module language.

See the CREATE MODULE Statement or the *Oracle Rdb Guide to SQL Programming* for information about stored module, stored routine, and stored routine parameter names.

2.2.13 Parameters, Routine Parameters, and SQL Variables

Parameters, routine parameters, and SQL variables are often used in value expressions (for information on value expression, see Section 2.6).

A **variable** is an identifier that represents a value that can change during the execution of a program. You can use SQL variables in multistatement procedures.

A **routine parameter** is a variable associated with a parameter of a routine that is used in a stored routine or an external routine. A **stored routine** refers to both stored procedures and stored functions defined using the CREATE MODULE statement. An **external routine** refers to both external procedures and external functions defined using the CREATE PROCEDURE and CREATE FUNCTION statements.

When you use SQL variables in multistatement procedures or when you use routine parameters, you do not use indicator variables. See Section 2.2.13.3 for more information about SQL variables in multistatement procedures and stored routine parameters. See Section 2.2.13.4 for more information about external routine parameters.

A **parameter** is an identifier declared in a host language program that is associated with an SQL statement. A parameter represents values that can change during the execution of a program. Many SQL data manipulation clauses that do not accept general value expressions require parameters. However, you cannot use parameters in data definition language statements.

You can use parameters in the following places:

- Interactive SQL

  In interactive SQL, you use the DECLARE Variable statement to declare the parameter. For more information about declaring parameters and variables, see DECLARE Variable Statement.

- SQL module language

  In programs that call SQL module procedures containing SQL statements, references to host language variables by SQL statements are indirect. The variable declared in the program is specified as a parameter in a host language call statement to a procedure in the SQL module. Parameters in such call statements are called **actual parameters**.

  In nonstored procedures, the SQL module procedure contains parameter declarations that correspond to the actual parameters in the calling program. Module parameters in those declarations are called **formal parameters**. The SQL statement in the module procedure uses the formal parameter name to refer indirectly to the actual parameter named in the host language call to the module procedure.

- Precompiled SQL

  In precompiled programs, SQL statements embedded in the program refer directly to the host language variable using it in the statement. The SQL precompiler supports only a subset of the declaration syntax for host languages. See Section 4.4 for more information. You can only use parameter names that conform to the rules of the host language.

- Dynamic SQL

  In dynamic SQL, dynamically executed SQL statements refer to parameters with **parameter markers** denoted by a question mark (?) in the statement string of PREPARE statements.

SQL statements use parameters for the following purposes:

- SQL retrieves data from the database and places it in parameters for use by a program.
- Parameters contain data generated by a program that SQL uses to update the database.
- Data manipulation statements can specify parameters in value expressions.
- Special-purpose parameters called indicator parameters indicate whether or not the value stored in a corresponding main parameter is null. (Indicator parameters are not used in stored routines.)

- SQL puts information about the success or failure of SQL statements in a parameter called SQLCODE that is either declared explicitly or as part of the SQL Communications Area (SQLCA) or in the SQLSTATE status parameter (ANSI/ISO SQL standard).

  See Appendix C for more information on SQLCODE and SQLSTATE.
- SQL and programs use a collection of parameters called the SQL Descriptor Areas (SQLDA and SQLDA2) to communicate information about dynamic SQL statements. See Appendix D for more information.

SQL statements cannot use parameters to refer to columns, tables, or views. For instance, if BADVAR is a host language variable that contains the name of a table in the database, the following statement is invalid:

```
EXEC SQL SELECT FIRST_NAME INTO :GOODVAR FROM :BADVAR END-EXEC
```

When you use the precompiler, module language, or dynamic SQL, display operations should use CAST or EXTRACT with CHAR host variables to convert date-time data from binary format when passing data to and from the database. For example:

```
EXEC SQL SELECT CAST(TBL_INT_H3 AS CHAR(4))
         INTO :string_var3
         FROM ALL_DATE_TABLE;
```

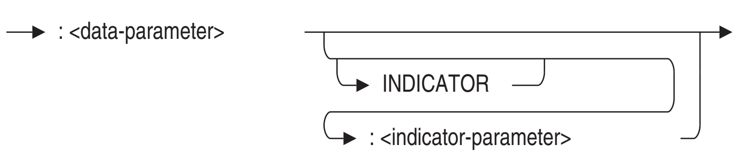
For more information about the CAST and EXTRACT functions, see Section 2.6.2.2 and Section 2.6.2.9, respectively.

Section 2.2.13.1 provides more information about parameters.

2.2.13.1 Data Parameters and Indicator Parameters

A **data parameter** contains the value that an SQL statement stores in, retrieves from, or compares to a column in the database. An **indicator parameter** specifies whether or not its associated data parameter was assigned a null value. You specify an indicator parameter after the data parameter. As for data parameters, the notation for referring to indicator parameters depends on the environment in which an SQL statement is issued.

parameter =



- If you set the dialect to SQL99 or another dialect that enforces the use of parameters or if you use a clause, such as PARAMETER COLONS, that enforces the use of parameters, all parameter names must begin with a colon. This rule applies to declarations and references of module language procedure parameters. If you do not use one of these dialects or clauses, no parameter name can begin with a colon. For more information, see SET DIALECT Statement and Section 3.2.

  The current default behavior is no colons are used. However, this default is deprecated syntax. In the future, colons will be the default because it allows processing of ANSI/ISO standard modules.
- In SQL statements to be dynamically executed, you refer to the data parameters and indicator parameters with a single parameter marker (?). SQL gets information about the parameters in EXECUTE or OPEN statements. These statements either provide an explicit list of data parameters and indicator parameters (using the notation for precompiled SQL or SQL modules as appropriate) or refer to the SQLDA that has fields that provide information about data parameters (SQLDATA) and indicator parameters (SQLIND).

Note

In SQL statement syntax diagrams, the parameter syntax element refers to any of the notations for data parameters and indicator parameters.

Oracle Rdb recommends that programs declare all indicator parameters as integers (signed longwords) in the host language program:

- COBOL: PIC S9(9) COMP

  To comply with the ANSI/ISO SQL standard, SQL also supports *sign leading separate* indicator variables in COBOL and the BINARY argument.

  BINARY is a synonym for COMP. For more information, see the supporting documentation for the COBOL language.

  COBOL: PIC S9(9) SIGN LEADING SEPARATE
- FORTRAN: INTEGER*4
- PL/I: BIN FIXED(31)
- C: int num2
- Ada: STANDARD.INTEGER

- BASIC: LONG (module language only)
- Pascal: [LONG] –MAXINT . . . +MAXINT

You declare indicator parameters as an array only when they are used with a reference to a host structure (see Section 2.2.13.2).

Table 2–6 summarizes when indicator parameters in nonstored procedures are necessary and how SQL treats null values.

Table 2–6 Indicator Parameters and Null Values

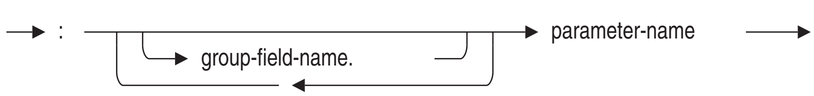
	Retrieval from Database		Storage into Database	
Nulls Allowed?	**Main Parameter**	**Indicator Parameter**	**Main Parameter**	**Indicator Parameter**
Nulls allowed; value is not null.	Set to value from database by SQL	Set to 0 or a positive value by SQL	Program must set to value to be stored	Program must set to 0 or a positive value
Nulls allowed; value is null.	Unchanged from previous value; program should disregard	Set to –1 by SQL	Ignored by SQL	Program must set to a negative value
Null values are not allowed.	Set to value from database by SQL	Not necessary; set to 0 or a positive value by SQL if present	Program must set to value to be stored	Not necessary; program must set to 0 or a positive value if present

2.2.13.2 Host Structures and Indicator Arrays

Host structures are host language parameters that correspond to group constructs or records in the languages that support such constructs. Use a host structure to refer to a list of host language variables with a single name. Once you define a host structure, you can refer to it in an embedded SQL statement or in an SQL module language procedure instead of listing the host language variables that comprise it.

Parameters can be qualified by group fields to any depth. The format of a qualified reference to a parameter in a group construct is:

qualified-parameter =



In addition, you can declare an indicator parameter for a host structure by defining a one-dimensional array of signed longword integers. This array provides indicator parameters for fields in the host structure and is called an **indicator array**. (Indicator arrays are also called indicator structures or indicator vectors.) Just as you append an indicator parameter to a data parameter, you can append the name of an indicator array to a host structure that represents several data parameters. Indicator arrays are the only way to specify indicator parameters for host structures.

You can refer to a host structure anywhere that SQL allows a list of parameters:

- VALUES clause of an INSERT statement
- Select lists
- IN predicates
- INTO clause of FETCH or singleton SELECT statements
- USING clause of OPEN or EXECUTE statements

You cannot use host structures in a stored routine or a multistatement procedure.

The following example shows the declarations in a COBOL program for a host structure and indicator array that correspond to the EMPLOYEES table in the personnel database. It also shows an embedded SQL INSERT statement that uses the host structure and indicator array.

```
 .
 .
 .
WORKING-STORAGE SECTION.
*
* Host structure declaration. A parameter to match
* each column being retrieved or stored is a subordinate
* field in the structure.
*
01 WS-EMP-REC.
        02 WS-EMP-ID            PIC X(5).
        02 WS-L-NAME            PIC X(14).
        02 WS-F-NAME            PIC X(10).
        02 WS-M-INIT            PIC X.
        02 WS-ADDRESS-1         PIC X(25).
        02 WS-CITY              PIC X(20).
        02 WS-STATE             PIC X(2).
        02 WS-POSTAL-CODE       PIC X(5).
        02 WS-SEX               PIC X.
        02 WS-BIRTH-DATE        SQL_DATE.
        02 WS-STATUS            PIC X.
```

```
*
* Indicator array for host structure WS-EMP-REC.
* EMP-REC-IND is the indicator when you refer to WS-EMP-REC.
*
01 WS-EMP-REC-IND.
        02 EMP-REC-IND OCCURS 11 TIMES PIC S9(9) COMP.
*
* Indicator declarations for references to individual parameters
* in WS-EMP-REC.  You cannot use a subscripted reference to the
* indicator array in such references, but must declare separate
* indicator parameters.
*
01 EMP-ID-IND                   PIC S9(9) COMP.
01 L-NAME-IND                   PIC S9(9) COMP.
01 F-NAME-IND                   PIC S9(9) COMP.
01 M-INIT-IND                   PIC S9(9) COMP.
01 ADDRESS-1-IND                PIC S9(9) COMP.
01 CITY-IND                     PIC S9(9) COMP.
01 STATE-IND                    PIC S9(9) COMP.
01 POSTAL-CODE-IND              PIC S9(9) COMP.
01 SEX-IND                      PIC S9(9) COMP.
01 BIRTH-DATE-IND               PIC S9(9) COMP.
01 STATUS-IND                   PIC S9(9) COMP.
 .
 .
 .
EXEC SQL
INSERT INTO EMPLOYEES VALUES (:WS-EMP-REC:EMP-REC-IND)
END-EXEC.
 .
 .
 .
```

You can also refer to a single parameter in a host structure. In FORTRAN, C, Pascal, and Ada, you must qualify the parameter name with all preceding group field names. In COBOL and PL/I, you need to qualify the parameter with group field names only if the name is ambiguous without such qualification.

Keep in mind the following notes about host structures and indicator arrays in embedded SQL statements:

- You must declare separate indicator parameters for each host language parameter in a structure to which you want to refer. For instance, in the preceding COBOL example's declaration of WS-EMP-REC and WS-EMP-REC-IND, one correct way to refer to the host structure and indicator parameters for the F-NAME field is:

  ```
  :WS-EMP-REC.WS-F-NAME:F-NAME-IND.
  ```

You cannot use subscripted references to individual elements of an indicator array as indicator parameters for individual parameters of a host structure. In the preceding COBOL declaration of WS-EMP-REC and WS-EMP-REC-IND, an SQL statement could not refer, for example, to the host structure and indicator parameters for the F-NAME field as:

```
:WS-EMP-REC.WS-F-NAME:EMP-REC-IND(3).
```

- COBOL and the SQL precompiler differ in how they interpret references to host structures. This difference can lead to a precompiler error message that may be confusing.

  COBOL interprets a reference to a host structure as a reference to a single parameter that has a text data type and the length of the concatenated subordinate fields in the structure. For example, COBOL interprets a reference to B-DATE in the following declaration as a reference to a single parameter that contains the values in the elementary fields of the structure:

```
01 B-DATE.
   02   CENTURY PIC XX.
   02   YEAR    PIC XX.
   02   MONTH   PIC XX.
   02   DAYW    PIC XX.
```

  However, SQL interprets a reference to a host structure as a reference to all the individual parameters that comprise it. An embedded SQL statement that refers to B-DATE must treat B-DATE as four separate host language parameters. For example, the following SQL statement embedded in the same program with the previous B-DATE declaration generates a precompiler error:

```
EXEC SQL
INSERT INTO TEMP_TABLE (BIRTHDAY) VALUES (:B-DATE)
END-EXEC.
* This statement will generate this precompiler error:
*
* %SQL-F-INVVALLIS,
* The value list must have as many items as the column list.
```

  You can work around this problem by declaring B-DATE as a single parameter, then using the COBOL REDEFINES clause to declare four parameters that refer to it, as follows:

```
01 B-DATE PIC X(9).
01 B-DATE-REDEF REDEFINES B-DATE.
   02   CENTURY PIC XX.
   02   YEAR    PIC XX.
   02   MONTH   PIC XX.
   02   DAYW    PIC XX.
```

COBOL host structures associated with VARCHAR or LONG VARCHAR columns are exceptions to the rule that SQL interprets references to host structures as separate references to the elementary fields that comprise them. For these host structures, SQL interprets the two elementary fields as a single parameter to or from which to assign a varying-text value. See Section 4.4.4 for details on declaring COBOL host structures for VARCHAR or LONG VARCHAR columns.

2.2.13.3 Multistatement Procedure Variables and Stored Routine Parameters

Multistatement procedure variables and stored routine parameters are often used in value expressions (see Section 2.6). A **variable** is an identifier that represents a value that can change during the execution of a program. You use SQL variables in multistatement procedures. A **stored routine parameter** is a variable associated with the parameters of a stored routine that you use in a stored procedure or stored function. A **stored routine** refers to both stored procedures and stored functions defined using the CREATE MODULE statement.

variable =

Variables in multistatement procedures and stored routine parameters follow the rules, such as case-sensitivity rules, associated with the encompassing module. That means:

- In embedded SQL, the variables follow the rules for the host language in which the program is written.
- In SQL module language programs, the variables follow the rules for the SQL interface.
- In stored routines, the variables follow the rules for the SQL interface.

Unlike data parameters, variables and stored routine parameters allow null values. Because of this, you cannot use indicator parameters with variables and stored routine parameters.

For more information about stored routine parameters, see CREATE MODULE Statement.

2.2.13.4 External Routine Parameters

An **external routine parameter** is a 3GL declaration that corresponds to an actual parameter in the calling program. These declarations are called **formal parameters**. 3GL or SQL statements in the external routine use the formal parameter name to refer indirectly to the calling programs actual parameters.

External routine parameters cannot represent null values.

2.2.14 Statement Names (Dynamic SQL Only)

Dynamic SQL lets programs accept or generate SQL statements at run time, in contrast to precompiled statements that must be embedded in the program before it is compiled. Unlike embedded statements, such dynamically executed SQL statements are not part of any source code but are created while the program is running. Dynamic SQL is useful when you cannot predict the type of SQL statement your program needs to process.

To handle dynamically executed SQL statements, programs use embedded PREPARE statements to assign a name to the SQL statement created at run time and to prepare it for execution. The EXECUTE, dynamic DECLARE CURSOR, and DESCRIBE statements refer to that assigned name. You cannot qualify prepared statement names.

Because they are prepared with embedded PREPARE statements, you can refer to dynamic statement names from programs only, not from interactive SQL.

2.2.15 Schema Names

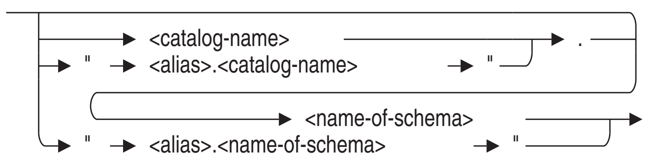
A **schema** consists of metadata definitions such as tables, views, domains, constraints, collating sequences, indexes, storage maps, triggers, and the privileges for each of these.

You name schemas in CREATE SCHEMA or CREATE DATABASE statements. You can also use schema names to qualify the names of other database elements such as tables, views, and columns.

Note

In syntax diagrams, the schema-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the schema in the CREATE statement. That is, in syntax diagrams, the schema-name is always defined as:

schema-name =



By default, each database that you create has only one schema. CREATE DATABASE Statement tells how to create a multischema database. The alias RDB$DBHANDLE represents the schema when you refer to definitions in a single-schema database or definitions in a multischema database without multischema naming enabled.

When you refer to definitions in a multischema database, you must follow multischema naming rules unless you disable multischema naming. In multischema naming:

- You must qualify definition names using the name of the schema that contains them. You cannot refer to a table and a view or two objects of the same type (such as two tables) with the same name unless they belong to different schemas.
- You may additionally qualify the names of objects in a multischema database with the alias and the catalog name.

  Whenever you qualify the object name with a catalog name, you must also specify the schema name, unless you want to use the default schema. Remember that the catalog name and alias combination or the schema name and alias combination must be enclosed within double quotation marks.
- If you prefer, you can qualify an object name in a multischema database with just an alias, provided you have set the default catalog and schema to the ones that you want to contain the object. Enclose the alias and object name pair within double quotation marks and separate them with a period.

If you omit the schema name when referring to objects in a multischema database, SQL uses a schema with the same name as the user identifier of the invoker as the default schema. You can use the SET SCHEMA statement to change the default schema.

The following example creates a table, QUARTERLY_TOTAL, in the schema RDB$SCHEMA in the catalog RDB$CATALOG of the multischema database with alias CORP.

```
SQL> ATTACH 'ALIAS CORP FILENAME corporate_data';
SQL> SET QUOTING RULES 'SQL99';
SQL> SET CATALOG 'RDB$CATALOG';
SQL> SET SCHEMA 'RDB$SCHEMA';
SQL> CREATE TABLE "CORP.QUARTERLY_TOTAL" (SALARY_AMOUNT_DOM CHAR);
SQL> SHOW TABLES;
User tables in database with alias CORP
     "CORP.ADMINISTRATION".ACCOUNTING.BUDGET
     "CORP.ADMINISTRATION".ACCOUNTING.DEPARTMENTS
     .
     .
     .
     "CORP.ADMINISTRATION".RECRUITING.RESUMES
     "CORP.RDB$CATALOG".RDB$SCHEMA.QUARTERLY_TOTAL
```

For more information about catalogs, see Section 2.2.3.

2.2.16 Storage Area Names

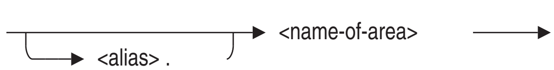
Storage areas are data and snapshot files that are associated with one or more tables in a multifile database. You name storage areas in CREATE STORAGE AREA clauses within CREATE DATABASE or IMPORT statements. The CREATE STORAGE MAP statements control which parts of which tables get stored in a particular storage area. In syntax diagrams, the syntax element area-name specifies that you supply the name of a storage area at that place in the statement. In CREATE STORAGE AREA clauses and in other SQL statements, the names you give to storage areas in the CREATE statement can be qualified by aliases.

You must use ASCII alphanumeric characters for the storage area name.

Note

In syntax diagrams, the area-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the storage area in the CREATE STORAGE AREA clause.

area-name =



2.2.17 Storage Map Names

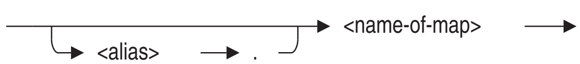
Storage maps control which parts of which tables get stored in a particular storage area in a multifile database. You name storage maps in CREATE STORAGE MAP statements. In syntax diagrams, the syntax element map-name specifies that you supply the name of a storage area at that place in the statement.

In CREATE STORAGE MAP and other SQL statements, the names you give to storage maps in the CREATE statement can be qualified by aliases.

Note

In syntax diagrams, the map-name syntax element refers to *either* the qualified *or* unqualified form of the name given to the storage map in the CREATE STORAGE MAP statement.

map-name =



2.2.18 Stored Names

The name that you specify for a data definition when you create it is called the **SQL name**. Each data definition also has a **stored name** that it is known by to Oracle Rdb.

You can give the same SQL name to two entities of the same type within different schemas of a multischema database. For example, you could create a table called EMPLOYEES in the schema DEPT1 and a second EMPLOYEES table in the schema DEPT2. For the first EMPLOYEES table created, SQL assigns a stored name that is the same as the SQL name. For subsequent EMPLOYEES tables, SQL generates a unique stored name by adding a serial number and truncating the name, if necessary.

Table 2–7 contrasts SQL and stored names for three definitions in a multischema database.

Table 2–7 Stored and SQL Names

For This SQL Name:	SQL Assigns This Stored Name:[1]
DEPT1.EMPLOYEES	EMPLOYEES
DEPT2.EMPLOYEES	EMPLOYEES1
DEPT3.EMPLOYEES	EMPLOYEES2

[1]This table assumes that the EMPLOYEES in DEPT1, DEPT2, and DEPT3 are created sequentially.

If you prefer to specify a stored name for a definition in a multischema database instead of relying on SQL to generate one, you can do so using the STORED NAME IS clause for any CREATE statement. You can only specify stored names for definitions in multischema databases.

SQL requires that, for each definition of a particular type, the SQL name must be unique within the schema, and the stored name must be unique within the database.

The stored name allows you to access multischema definitions using interfaces, such as Oracle RMU, the Oracle Rdb management utility, that do not recognize multiple schemas in one database. You can access multischema definitions by their stored names if you disable multischema naming using the MULTISCHEMA IS OFF clause in the ATTACH or DECLARE ALIAS statement.

2.2.19 Table and View Names

You name tables and views in CREATE TABLE and CREATE VIEW statements. In those and other SQL statements, the names you give to tables and views in CREATE statements can be qualified by aliases and can themselves qualify column names.

If your database has the multischema option enabled, you can also qualify table and view names by schema and catalog names, or by the alias. You must use double quotation marks to surround the alias and table name pair and have set your dialect to the ANSI/ISO SQL standard or use the ANSI/ISO SQL standard quoting rules. See the SET DIALECT Statement and the SET QUOTING RULES Statement for more information about dialects and quoting rules. The following are valid names for the EMPLOYEES table in the database with alias CORP, catalog ADMINISTRATION, and schema PERSONNEL:

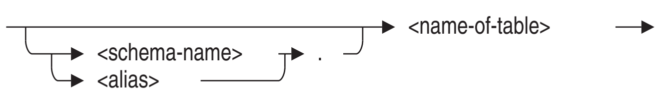
- "CORP.ADMINISTRATION".PERSONNEL.EMPLOYEES

• "CORP.EMPLOYEES"

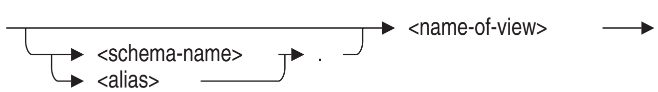
Note

In syntax diagrams, the table-name and view-name syntax elements refer to *either* the qualified *or* unqualified form of the names given to the table or view in the CREATE statement. That is, in syntax diagrams, table-name and view-name are always defined as:

table-name =



view-name =



You must qualify table names and view names with an alias if they are not in the default database. The following example shows the error that SQL generates if you try to use an unqualified table name to refer to a database previously declared with an alias:

```
SQL> ATTACH 'ALIAS PERS PATHNAME personnel';
SQL> SET QUOTING RULES 'SQL92';
SQL> SELECT * FROM EMPLOYEES;
%SQL-F-NODEFDB, There is no default database
SQL> -- This statement will work:
SQL> SELECT * FROM "PERS.EMPLOYEES";
```

The system default catalog is RDB$CATALOG. The system default schema is the user name. These defaults can be set in the SQL module header, the precompiler context file, or interactively by the SET statement. In a multischema database, you must qualify table names and view names with a catalog name if they are not in the default catalog, and also with a schema name if the tables and views are not in the default schema. The error message shows that the default schema is set to the user name LUFKIN.

```
SQL> SELECT * FROM "CORP.EMPLOYEES";
%SQL-F-SCHNOTDEF, Schema "CORP.RDB$CATALOG".LUFKIN is not defined
SQL> SELECT * FROM "CORP.ADMINISTRATION".PERSONNEL.EMPLOYEES;
 EMPLOYEE_ID   LAST_NAME             FIRST_NAME            MIDDLE_INITIAL
   ADDRESS_DATA_1             ADDRESS_DATA_2             CITY
      STATE   ZIP_CODE   SEX    BIRTHDAY    STATUS
 00164        Toliver               Alvin                   A
   146 Parnell Place                                     Chocorua
      NH      03817      M      1947-03-28   1

 00165        Smith                 Terry                   D
   120 Tenby Dr.                                         Chocorua
      NH      03817      M      1954-05-15   2
SQL> --
SQL> -- By changing the default catalog from RDB$CATALOG to the
SQL> -- catalog containing EMPLOYEES, you can avoid specifying the
SQL> -- catalog name.
SQL> --
SQL> SET CATALOG ADMINISTRATION;
SQL> SELECT * FROM "CORP.PERSONNEL".EMPLOYEES;
 EMPLOYEE_ID   LAST_NAME             FIRST_NAME            MIDDLE_INITIAL
   ADDRESS_DATA_1             ADDRESS_DATA_2             CITY
      STATE   ZIP_CODE   SEX    BIRTHDAY    STATUS
 00164        Toliver               Alvin                   A
   146 Parnell Place                                     Chocorua
      NH      03817      M      1947-03-28   1

 00165        Smith                 Terry                   D
   120 Tenby Dr.                                         Chocorua
      NH      03817      M      1954-05-15   2
```

The next example copies data from one database to another. Because the example declares both databases using aliases, references to tables in either database must be qualified by the alias for their respective database. In this case, the table names for both databases are the same, and aliases help distinguish a table in the target database from a table of the same name in the source database. The example uses an empty copy of the personnel sample database called temp and follows this sequence:

```
SQL> -- Use the alias empty for the temp database:
SQL> --
SQL> ATTACH 'ALIAS empty PATHNAME temp'; 1
SQL> --
SQL> -- Use the alias pers for the personnel database:
SQL> --
SQL> ATTACH 'ALIAS pers PATHNAME personnel'; 1
SQL> --
```

```
SQL> -- Now declare a transaction, using the aliases to allow copying
SQL> -- from the personnel database to the temp database:
SQL> --
SQL> DECLARE TRANSACTION ON empty USING (READ WRITE)
cont>                 AND ON pers USING (READ ONLY); 2
SQL> --
SQL> -- Finally, use an INSERT statement to copy data from the
SQL> -- personnel database into the empty table, qualifying the table
SQL> -- with the aliases:
SQL> --
SQL> INSERT INTO empty.employees 3
cont>   SELECT * FROM pers.employees;
```

1 The ATTACH statements specify aliases of EMPTY and PERS.

2 The DECLARE TRANSACTION statement uses the aliases to include both databases in a single transaction.

3 The INSERT statement uses those aliases to distinguish between the EMPLOYEES table in the personnel database and the EMPLOYEES table in the temp database.

For an example of using table names to qualify column names, see Section 2.2.4.

2.2.20 Trigger Names

You name a trigger in the CREATE TRIGGER statement. A trigger name must be unique within a schema of a multischema database or unique within a nonmultischema database.

A **trigger** defines the actions to occur before or after a specified table is updated (by a write operation such as an INSERT, DELETE, or UPDATE statement). A trigger can be thought of as a rule on a single table, which takes effect at a specific time for a particular type of update and causes one or more triggered actions to be performed.

With triggers, you can define useful actions such as:

- Cascading deletes

  Deleting a row from one table causes additional rows to be deleted from other tables that are related to the first table by key values.

- Cascading updates

  Updating a row in one table causes additional rows to be updated in other tables that are related to the first table by key values. These updates are usually limited to the key values themselves.

- Summation updates

Updating a row from one table causes a value in a row of another table to be updated by being increased or decreased.

- Hidden deletes

  Causing rows to be deleted from a table by moving them to a parallel table that is not used by the database.

- Audit log

  Records when and by whom a row is inserted, updated, or deleted.

2.3 Data Types

When you define new columns of a table in the CREATE TABLE or ALTER TABLE statements, you must specify a data type for the column. The data type of a column controls how SQL interprets and stores values for that column. All value expressions (functions, parameters, and literals) have associated data types.

Table 2–8 lists the SQL data type keywords and the underlying OpenVMS data types.

Table 2–8 Comparison of SQL Keywords with OpenVMS Data Types

SQL Keywords	OpenVMS Data Types
CHAR (n)	Character string (DSC$K_DTYPE_T)
CHAR (n), qualified by character set	Character string (DSC$K_DTYPE_T)
NCHAR (n)	Character string (DSC$K_DTYPE_T)
VARCHAR (n)	Varying character string (DSC$K_DTYPE_VT)
VARCHAR (n), qualified by character set	Varying character string (DSC$K_DTYPE_VT)
NCHAR VARYING(n)	Varying character string (DSC$K_DTYPE_VT)
LONG VARCHAR	Varying character string (DSC$K_DTYPE_VT)
TINYINT [(n)][1]	Signed byte integer (DSC$K_DTYPE_B)
SMALLINT [(n)][1]	Signed word integer (DSC$K_DTYPE_W)

[1]Scale factors (n) in SQL integer data types are equivalent to negative scale factors in Oracle Rdb integer data types. SQL does not support Oracle Rdb positive scale factors.

(continued on next page)

Table 2–8 (Cont.) Comparison of SQL Keywords with OpenVMS Data Types

SQL Keywords	OpenVMS Data Types
INTEGER [(n)][1]	Signed longword integer (DSC$K_DTYPE_L)
QUADWORD [(n)][1,2]	Signed quadword integer (DSC$K_DTYPE_Q)
BIGINT [(n)][1,2]	Signed quadword integer (DSC$K_DTYPE_Q)
DECIMAL [(n[,n])][3]	Packed decimal string (DSC$K_DTYPE_P)
NUMERIC [(n[,n])][3]	Numeric string, left separate sign (DSC$K_DTYPE_NL)
FLOAT [(n)]	Single-precision (F-floating) or double-precision (G-floating) floating-point number, depending on n (DSC$K_DTYPE_F or DSC$K_DTYPE_G)
REAL	Single-precision floating-point number (DSC$K_DTYPE_F)
DOUBLE PRECISION[4]	Double-precision floating-point number: G-floating (DSC$K_DTYPE_G)[3]
DATE	DATE VMS (default DATE) is DSC$K_DTYPE_ADT, DATE ANSI is internal to Oracle Rdb
TIME	Internal to Oracle Rdb
TIMESTAMP	Internal to Oracle Rdb
INTERVAL	Internal to Oracle Rdb
LIST OF BYTE VARYING	–
BYTE VARYING	–[5]

[1]Scale factors (n) in SQL integer data types are equivalent to negative scale factors in Oracle Rdb integer data types. SQL does not support Oracle Rdb positive scale factors.

[2]Oracle Rdb recommends that you use the keyword BIGINT in place of QUADWORD.

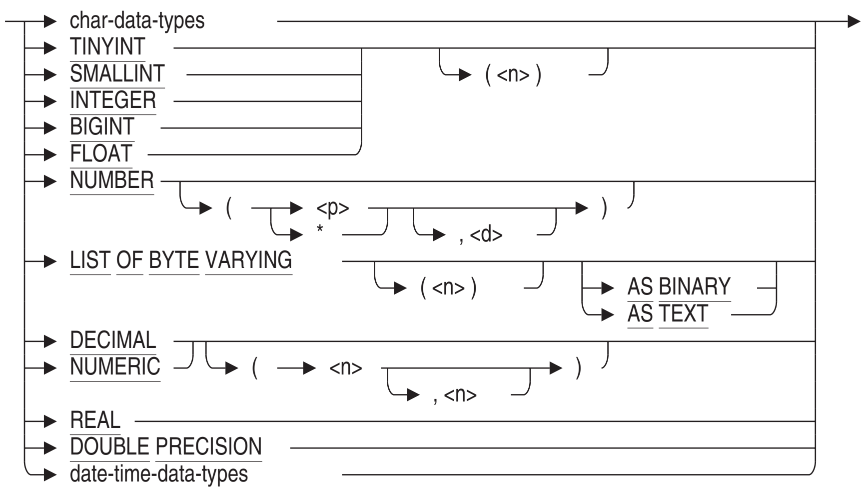
[3]Because the DECIMAL and NUMERIC data types are not supported, SQL creates integer or floating-point columns in the database when it encounters DECIMAL or NUMERIC in table definitions. However, SQL converts between integer, character, or floating-point values in database columns and numeric string values in procedure parameters and host language variables.

[4]SQL converts from G-floating values in the database to a D-floating representation for host languages that do not support the G-floating data type.

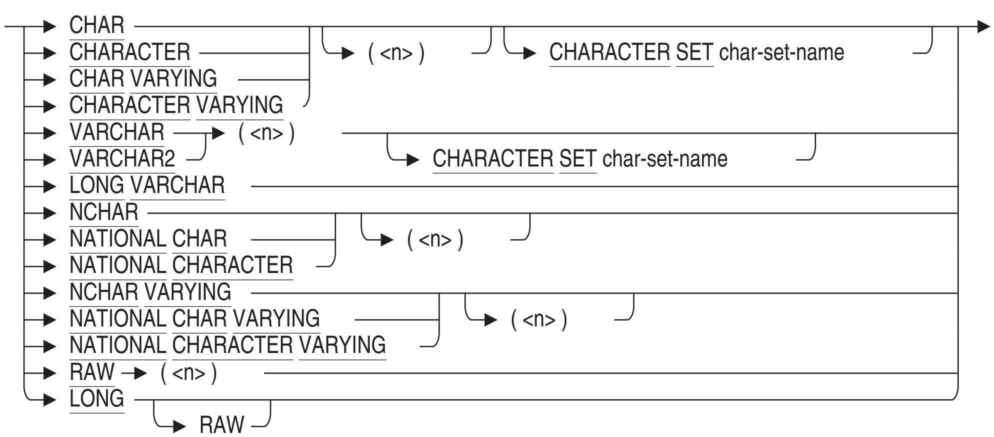
[5]The BYTE VARYING data type is a string of unsigned 8-bit bytes. It is currently only valid as the format for an SQL LIST segment but is reserved for future use.

Use the following format when you specify a data type:

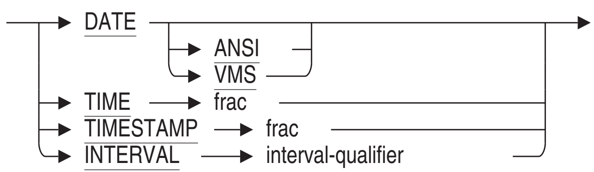
data-type =



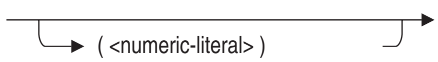
char-data-types =



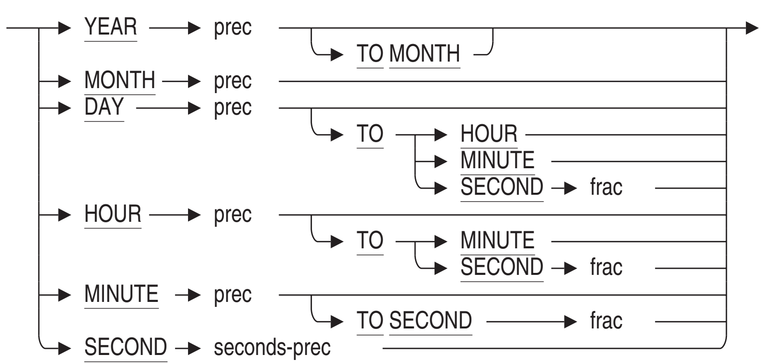
date-time-data-types =



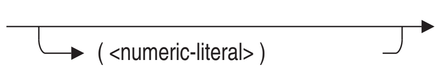
frac =



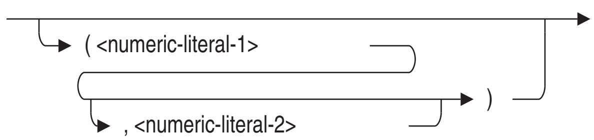
interval-qualifier =



prec =



seconds-prec =



The following sections describe character, DECIMAL and NUMERIC, fixed- and floating-point numeric, date-time, LIST OF BYTE VARYING data types, and rules for converting between data types.

2.3.1 Character Data Types

SQL supports the following character data types:

- CHAR

  This data type specifies that the column is a fixed-length sequence of octets or characters. It indicates the number of octets or characters in the column with an unsigned integer (*n*). (See Table 2–2 for a list of the number of octets used by characters in the supported character sets). The maximum size for *n* is 65,271 octets. For characters, the maximum size for *n* is 65,271 divided by the maximum number of octets per character. For example, the Kanji character set uses a maximum of 2 octets. Therefore, *n* is 65,271/2 or 32,635 characters. If you omit *n*, SQL creates a 1-octet or 1-character column.

- CHAR or CHARACTER qualified by the keywords CHARACTER SET and the character set name

  This data type has the same characteristics as CHAR, except that the character set is that specified in the CHARACTER SET clause. For a list of the character set names, see Section 2.1.

- NCHAR, NATIONAL CHAR, or NATIONAL CHARACTER

  This national character data type has the same characteristics as CHAR, except that the character set is that specified as the national character set.

- VARCHAR or CHARACTER VARYING

  This data type specifies that the column is a varying-length sequence of octets or characters. It indicates the maximum number of octets or characters in the column with an unsigned integer (*n*). The maximum size for *n* is 65,269 octets. For characters, the maximum size for *n* is 65,269 divided by the maximum number of octets per character. For example, the

Kanji character set uses a maximum of 2 octets. Therefore, *n* is 65,269/2 or 32,634 characters.

In addition to the VARCHAR2 synonyms, the CHARACTER VARYING and CHAR VARYING data type keywords are supported by Oracle Rdb as synonyms for VARCHAR, in compliance with the ANSI/ISO SQL standard.

- VARCHAR qualified by the keywords CHARACTER SET and the character set name

  This data type has the same characteristics as VARCHAR, except that the character set is that specified in the CHARACTER SET clause. For a list of the character set names, see Section 2.1.

- NCHAR VARYING, NATIONAL CHAR VARYING, or NATIONAL CHARACTER VARYING

  This national character set data type has the same characteristics as VARCHAR, except that the character set is that specified as the national character set.

- LONG VARCHAR

  This data type specifies that the column is a varying-length sequence of octets or characters with a maximum number of 16,383 octets. For characters, the maximum size for *n* is 16,383 divided by the maximum number of octets per character. For example, the Kanji character set uses a maximum of 2 octets. Therefore, *n* is 16,383/2 or 8,191 characters. The LONG VARCHAR data type is equivalent to specifying VARCHAR (16383).

- RAW is a synonym for VARCHAR. It always has a character set of UNSPECIFIED (that is, the CHARACTER SET clause is not permitted) and so can be assigned data from any other character set in the database. This allows data from any character set to be assigned to a column, parameter, or variable of this type. If you specify a length, the value can be 0 to 65535. Actual usage might be limited by available space in a table row, but the full length can be used by variable and parameter definitions (as is true for CHAR, VARCHAR, and VARCHAR2 data types).

- LONG is a synonym for LIST OF BYTE VARYING AS TEXT.

- LONG RAW is a synonym for LIST OF BYTE VARYING AS BINARY.

For each data type, the length of each character can be one or more octets, depending upon the character set. By default, the length of a character data type is octets. To specify the length in characters, use the SET DIALECT or SET CHARACTER LENGTH statements.

If you do not qualify the data type with a character set, SQL considers the column to be of the character set specified as the database default character set. If you do not specify a default character set for the database, SQL considers the column to be the DEC_MCS character set.

You cannot use text values in arithmetic expressions—whether they are literals, stored in parameters, or literals stored in table columns.

```
SQL> SELECT EMPLOYEE_ID + 1 FROM EMPLOYEES;
%SQL-F-UNSSTRXPR, Unsupported string expression
```

Note

By default, SQL treats C language character strings as null-terminated strings. If you want to create a C application to manipulate binary input:

- Use the $SQL_VARCHAR data type with the SQL C precompiler.
- Use SQL module language with GENERAL as the language qualifier.
- Use SQL module language and the repository with FIXED as the character string interpretation option.

2.3.1.1 Calculating the Maximum Length of a CHAR or VARCHAR Column

All SQL data types take up a fixed amount of room in a database row. Most are predetermined in size. For example, the BIGINT data type requires 8 octets for storage. However, the CHAR, VARCHAR, and the NATIONAL CHARACTER equivalent data types allow you to specify the storage size in characters.

Oracle Rdb restricts a stored row to 65,272 octets which limits the number of columns and the associated data type sizes for the table.

There is also a variable overhead for each table definition which is a varying number of octets to represent the NULL flags (one flag for each column). The larger the number of columns in the table, the larger the NULL bit vector (which is stored as a whole number of octets). For each eight columns in the table, a single octet is used to store the NULL bit vector.

Note

Each VARCHAR, NATIONAL CHARACTER VARYING (or equivalent syntax) column requires two additional octets in which to save the actual length.

The maximum length for a CHAR or VARCHAR column is controlled by the amount of free space available to store the new column. For example, it is possible to create a table with a single CHAR(65271) column.

```
SQL> CREATE TABLE T1 (A CHAR(65271));
```

However, if you define an additional BIGINT column, the maximum CHAR length is reduced to 65,263 octets (65,271 minus 8 octets).

```
SQL> CREATE TABLE T1 (A CHAR(65271), B BIGINT);
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RECMAXEXC, relation T1 definition exceeds data limit
```

```
SQL> CREATE TABLE T1 (A CHAR(65263), B BIGINT);
```

The maximum character length is dependent upon the types and number of columns in the table.

2.3.2 Date-Time Data Types

SQL provides four data types for expressing dates and times, hereafter called date-time data types. The DATE, TIME, and TIMESTAMP data types refer to calendar date and clock time. The INTERVAL data type is a relative date-time data type that refers to the duration between two date-time values.

The date-time data types are:

- DATE

  You can qualify DATE with two keywords:

  - DATE ANSI specifies a DATE containing Year To Day.
  - DATE VMS specifies a timestamp containing YEAR TO SECOND.

  If you do not qualify the DATE data type, it is interpreted as DATE VMS when creating columns in a table. When you issue an INSERT or SELECT statement, you must qualify the DATE data type. The DATE VMS data type cannot be used in date-time arithmetic.

  You can change DATE to DATE ANSI 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. 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 create a database definition.

  For information on the format of the DATE data type, see Section 2.4.3.

- TIME

  Contains the fields HOUR, MINUTE, and SECOND. You can specify a fractional-seconds precision following TIME. The fractional-seconds precision, shown in the syntax diagram in Section 2.3 as *frac*, is a number between 0 and 2 that represents the number of digits taken up by fractions of a second. If you specify TIME without a fractional-seconds precision, it defaults to TIME(0).

- TIMESTAMP

  Contains the fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. You can specify a fractional-seconds precision following TIMESTAMP. The fractional-seconds precision, shown in the syntax diagram in Section 2.3 as *frac*, is a number between 0 and 2 that represent the number of digits taken up by fractions of a second. If you specify TIMESTAMP without a fractional-seconds precision, it defaults to TIMESTAMP(2), hundredths of a second.

- INTERVAL

  Specifies the difference between two date-time data types.

  To qualify which interval in which you want an interval calculation expressed, SQL provides two categories of intervals, each with its own set of interval qualifiers, as Table 2–9 shows.

Table 2–9 Interval Qualifiers

Interval Category	Interval Qualifiers
YEAR-MONTH	YEAR
	YEAR TO MONTH
	MONTH
DAY-TIME	DAY
	DAY TO HOUR
	DAY TO MINUTE
	DAY TO SECOND

(continued on next page)

Table 2–9 (Cont.) Interval Qualifiers

Interval Category	Interval Qualifiers
	HOUR
	HOUR TO MINUTE
	HOUR TO SECOND
	MINUTE
	MINUTE TO SECOND
	SECOND

When formatting intervals, Oracle Rdb needs to know how many digits to expect in the leading field. The minimum value is 1 digit in the leading field, and the maximum width is 9 digits. The interval leading-field precision is shown in the syntax diagram in Section 2.3 as *prec*. If unspecified, the interval leading-field precision defaults to 2.

In the following example, the HOURS_WORKED column is computed from two TIMESTAMP columns. The leading-field precision for HOUR and SECOND interval qualifiers default to HOURS(2) and SECONDS(2).

```
SQL> CREATE TABLE ACCOUNTING.DAILY_HOURS
cont>   (EMPLOYEE_ID  CHAR(5),
cont>    START_TIME   TIMESTAMP,
cont>    END_TIME     TIMESTAMP,
cont>    HOURS_WORKED
cont>        COMPUTED BY (END_TIME - START_TIME) HOUR TO SECOND
cont>    );
SQL> --
SQL> -- Now show the columns in the table - note default precisions
SQL> --
SQL> SHOW TABLE (COLUMNS) ACCOUNTING.DAILY_HOURS;
Information for table DAILY_HOURS
```

```
Columns for table DAILY_HOURS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
EMPLOYEE_ID                     CHAR(5)
START_TIME                      TIMESTAMP(2)
END_TIME                        TIMESTAMP(2)
HOURS_WORKED                    INTERVAL
                                 HOUR (2) TO SECOND (2)
 Computed:       BY (END_TIME - START_TIME) HOUR TO SECOND
SQL> --
SQL> -- Output shows the two-digit precision in HOUR and SECOND qualifiers
SQL> --
SQL> SELECT EMPLOYEE_ID, HOURS_WORKED FROM ACCOUNTING.DAILY_HOURS;
 EMPLOYEE_ID   HOURS_WORKED
 00415          09:44:36.85
 00415          10:16:21.25
 00415          10:30:17.57
 .
 .
 .
```

The fractional-seconds precision, shown in the syntax diagram in Section 2.3 as *frac*, represents the number of decimal digits after the decimal point for the SECOND field. This number represents fractions of a second. The fractional-seconds precision must be between 0 and 2. If unspecified, the fractional-seconds precision defaults to 2 for the SECOND field.

The INTERVAL qualifier must range from a higher to a lower date field. The order of significance for the date-time fields is (from highest to lowest) YEAR, MONTH, DAY, HOUR, MINUTE, SECOND. Table 2–10 and Table 2–11 show the fields that intervals can contain.

Table 2–10 Fields in Year-Month INTERVAL Columns

Keyword	Meaning	Valid Range
YEAR	Years	Signed value
MONTH	Months	Signed value, constrained to –11 . . . 11 for YEAR TO MONTH interval

If you specify only MONTH for the interval, SQL calculates the month value as $Y \times 12 + M$, where Y and M represent the year and month stored internally as (Y,M).

Table 2–11 Fields in Day-Time INTERVAL Columns

		Valid Range	
Keyword	**Meaning**	**If Leading-Field**	**Otherwise**
DAY	Days[1]	–3649634 . . . 3649634	(same)
HOUR	Hours[2]	–87591216 . . . 87591216	0 . . . 23
MINUTE	Minutes[3]	–999999999 . . . 999999999	0 . . . 59
SECOND	Seconds[4]	–21474836.47 . . . 21474836.47	0 . . . 59.99

[1]This value is approximately the number of days in 9999 years.

[2]HOUR limit is derived by multiplying maximum DAY by 24.

[3]This value is constrained by the maximum interval leading-field precision.

[4]This value is constrained by the maximum value that can be stored in a INTEGER(2) scale –2.

If you specify a subset of the day-time fields, SQL adds up the values from the most significant fields into an appropriate value for the highest leading-field specified.

SQL truncates less significant fields that are not specified in the interval qualifier. For example, assume that the interval is stored internally as (D,H,M,S). If you specify HOUR TO MINUTE in the interval qualifier, then SQL sets HOUR to $D \times 24 + H$, sets MINUTE to M, and truncates the SECOND field.

You can use date-time variables and constants in arithmetic expressions. The list of valid operators appears in Table 2–27. For more information about date-time arithmetic, see the *Oracle Rdb Introduction to SQL*.

For information on the compile-time translation of the YESTERDAY, TODAY, and TOMORROW character string literals, see Section 2.4.2.

Example 2–1 shows how to use several of these date-time data types.

Example 2–1 Using Date-Time Data Types

```
SQL> -- Create a simple table with a variety of data types.  Note the use
SQL> -- of date-time literals for the DEFAULT and CHECK clauses.
SQL> --
SQL> CREATE TABLE DATE_TEST
cont>   (A DATE VMS,
cont>    B DATE ANSI,
cont>    C TIME(0)
cont>      DEFAULT TIME '06:00:00',
cont>    D TIMESTAMP(2),
cont>    E INTERVAL YEAR(4)
cont>      CHECK(E > INTERVAL '10' YEAR)
cont>      NOT DEFERRABLE,
cont>    F INTERVAL DAY(3) TO MINUTE,
cont>    G CHAR(16));
SQL> --
SQL> -- Literal dates are represented as TEXT literals.  On OpenVMS,
SQL> -- the date format is controlled by the LIB$DT_INPUT_FORMAT
SQL> -- and SYS$LANGUAGE logical names.  For more information about
SQL> -- these logical names, see the OpenVMS documentation for the
SQL> -- Run-Time Library.
SQL> --
SQL> INSERT INTO DATE_TEST (A) VALUE ('2-APR-1957');
1 row inserted
SQL> SET LANGUAGE SPANISH
SQL> INSERT INTO DATE_TEST (A) VALUE ('2-abr-1957');
1 row inserted
SQL> SET LANGUAGE ENGLISH
SQL> SELECT A FROM DATE_TEST;
 A
  2-APR-1957 00:00:00.00
  2-APR-1957 00:00:00.00
2 rows selected
SQL> --
```

(continued on next page)

Example 2–1 (Cont.) Using Date-Time Data Types

```
SQL> -- The ANSI/ISO SQL standard specifies that only date-time literals
SQL> -- can be assigned to date-time columns (for example, DATE, TIME,
SQL> -- TIMESTAMP, and INTERVAL).  These date-time literals are used
SQL> -- in INSERT, SELECT, and UPDATE statements and in CREATE and ALTER
SQL> -- statements.
SQL> --
SQL> INSERT INTO DATE_TEST (B) VALUE (DATE '1993-2-23');
1 row inserted
SQL> INSERT INTO DATE_TEST (C) VALUE (TIME '12:20:00');
1 row inserted
SQL> INSERT INTO DATE_TEST (D) VALUE (TIMESTAMP '1993-2-23 12:20:00.00');
1 row inserted
SQL> INSERT INTO DATE_TEST (E) VALUE (INTERVAL '35' YEAR(4));
1 row inserted
SQL> INSERT INTO DATE_TEST (F) VALUE (INTERVAL '365:10:21' DAY(3) TO
MINUTE);
1 row inserted
SQL> --
SQL> -- DATE VMS columns can have associated edit strings defined
SQL> -- for them.  However, ANSI/ISO date-time values always print
SQL> -- in ANSI/ISO format unless you cast them to DATE VMS.
SQL> --
SQL> SELECT D, CAST(D AS DATE VMS)
cont>   FROM DATE_TEST
cont>   WHERE D IS NOT NULL;
 D
 1993-02-23 12:20:00.00   23-FEB-1993 12:20:00.00
1 row selected
SQL> --
```

(continued on next page)

Example 2–1 (Cont.) Using Date-Time Data Types

```
SQL> -- Oracle Rdb also supports another internal format for
SQL> -- DATE VMS.  This format is similar to the TIMESTAMP format
SQL> -- except that the punctuation is omitted.  This example assigns
SQL> -- the CHAR column (A) to the DATE VMS (G) column and then displays
SQL> -- the result.  In an application, the CHAR column could be a CHAR
SQL> -- host variable or module language parameter.
SQL> --
SQL> INSERT INTO DATE_TEST (G) VALUE ('1957020100000000');
1 row inserted
SQL> UPDATE DATE_TEST
cont>   SET A=G
cont>   WHERE G IS NOT NULL;
1 row updated
SQL> SELECT A, G
cont>   FROM DATE_TEST
cont>   WHERE G IS NOT NULL;
 A                         G
  1-FEB-1957 00:00:00.00   1957020100000000
1 row selected
SQL> ROLLBACK;
```

2.3.3 DECIMAL and NUMERIC Data Types

SQL provides limited support for the packed decimal (DECIMAL) and signed numeric (NUMERIC) data types:

- Conversion to integer or floating point in column definitions

  Because the databases that underlie SQL may not support these data types, if you specify the DECIMAL or NUMERIC data type for a column, SQL generates a warning message and creates the column with a data type that depends on the precision argument specified. For example:

  ```
  SQL> CREATE TABLE T (C DECIMAL(3));
  %SQL-I-NO_DECIMAL, C is being converted from DECIMAL to SMALLINT.
  ```

  Following is a list of the data types to which SQL converts:

  – DECIMAL(1) through DECIMAL(4) are converted to SMALLINT.

    NUMERIC(1) through NUMERIC(4) are converted to SMALLINT.

  – DECIMAL(5) (default for DECIMAL) through DECIMAL(9) are converted to INTEGER.

    NUMERIC(5) (default for NUMERIC) through NUMERIC(9) are converted to INTEGER.

  – DECIMAL(10) through DECIMAL(18) are converted to BIGINT.

NUMERIC(10) through NUMERIC(18) are converted to BIGINT.

– DECIMAL(19) and larger are converted to FLOAT.

  NUMERIC(19) and larger are converted to FLOAT.

- Conversion to packed decimal or signed numeric data types in formal parameters or host language parameters

  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 converts between the data types of values in the database and the DECIMAL or NUMERIC representation specified for corresponding parameters and host language parameters.

  Prior to Oracle Rdb V6.0, SQL allowed you to insert a value into a column that exceeded the precision specified. This behavior is maintained for databases created prior to Oracle Rdb V6.0.

  To comply with the ANSI/ISO SQL standard, Oracle Rdb V6.0 and higher generates an error message if you attempt to exceed the precision specified. For example:

```
SQL> INSERT INTO T (C) VALUE (9999);
%RDB-E-VALOUTRANGE, value outside the specified precision (3) for
column "C"
```

2.3.4 NUMBER Data Type

Oracle Rdb supports the NUMBER data type for compatibility with Oracle servers but has these differences: the Oracle server supports up to 38 digits of precision, and scale is restricted to between -84 and 127; Oracle Rdb release 7.1 supports 18 digits, and the scale is -128 to 127.

The NUMBER data type has these features:

- NUMBER

  If no precision is provided, then this maps to DOUBLE PRECISION.

- NUMBER (p) or NUMBER (p, d), where p is precision and d is fractional precision

  When the precision (p) is specified, it is used to map to an integer type, or DOUBLE PRECISION if the precision is greater than 18.

  $p <= 2$, maps to TINYINT
  $2 < p <= 4$, maps to SMALLINT
  $4 < p <= 9$, maps to INTEGER
  $9 < p <= 18$, maps to BIGINT
  $p > 18$ maps to DOUBLE PRECISION

- NUMBER (*) or NUMBER (*, d)

  The asterisk (*) is shorthand for the largest scaled binary value. For Oracle Rdb release 7.1, this is equivalent to specifying 18 digits, and selects a BIGINT mapping.

- NUMBER(p, -d)

  NUMBER allows *d*, the fractional precision, to be negative. If the scale is negative, the data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

2.3.5 Fixed-Point Numeric Data Types

SQL provides four fixed-point numeric data types: TINYINT, SMALLINT, INTEGER, and BIGINT. In all four, you can specify an optional unsigned integer (*n*). The integer is a scale factor that indicates the number of places to the right of the decimal point.

The scale factor must be an integer in the range from 0 to 127. If you do not specify *n*, the default is 0 (with no places to the right of the decimal point).

- TINYINT

  Specifies that the column is a signed byte. (A byte is 8 contiguous bits.) The TINYINT data type can store a range of values from –128 through 127.

- SMALLINT

  Specifies that the column is a signed 16-bit word. The SMALLINT data type can store a range of values from –32,768 to 32,767.

- INTEGER

  Specifies that the column is a signed 32-bit longword. The INTEGER data type can store a range of values from –2**31 to (2**31) –1.

- BIGINT

  Specifies that the column is a signed 64-bit quadword. The BIGINT data type can store a range of values from –2**63 to (2**63) –1.

2.3.6 Floating-Point Numeric Data Types

SQL provides three floating-point numeric data types:

- FLOAT

  Specifies that the column is a 32-bit (REAL) or 64-bit (DOUBLE PRECISION) floating-point number, depending on the precision indicated in the positive integer (*n*). If *n* is less than 25, FLOAT specifies a 32-bit floating-point number. If *n* is 25 or greater, FLOAT specifies a 64-bit floating-point number.

  The maximum value for *n* is 53. If FLOAT does not include *n*, it specifies a 64-bit floating-point number.

- REAL

  Specifies that the column is a 32-bit floating-point number with precision to 24 binary digits.

- DOUBLE PRECISION

  Specifies that the column is a 64-bit floating-point number with precision to 53 binary digits.

2.3.7 LIST OF BYTE VARYING Data Type

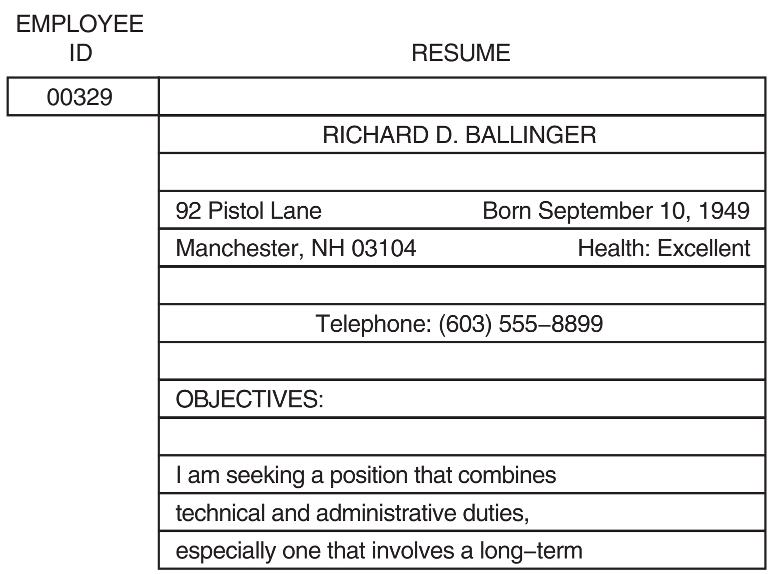
The LIST OF BYTE VARYING data type is designed to handle large data objects with a segmented internal structure. The LIST OF BYTE VARYING data type is equivalent to a:

- Segmented string
- Binary large object (BLOB) (certain industry implementations)
- LIST OF VARBYTE (alternate name in SQL syntax)

An object of the LIST OF BYTE VARYING data type is usually referred to as a list. A **list** is a linked list of data segments, with each segment stored on a separate page.

An example of a list can be seen in the RESUMES table of the sample personnel database. The RESUMES table contains two columns: a list column called RESUME and a character column called EMPLOYEE_ID. Figure 2–2 shows a conceptual diagram of the RESUMES table.

Figure 2–2 Table with a List Column



ZK–7548–GE

In a list, you can store unstructured data such as large amounts of text, long strings of binary input from a data collecting device, or graphics data. Any data type can be stored and retrieved from a list. The data is stored in unstructured bytes. For example, you can store character data in a list and then interpret it as hexadecimal data. Except for the length of the segments, Oracle Rdb does not know anything about the type of data contained in a list.

There is no limit on the number of segments within a list.

Each segment stored on a page is referenced by the line index structure, which uses a word offset and a word length. The page structure imposes a segment size limit of 65,535 unsigned bytes.

Use an unsigned integer (*n*) to specify the number of octets (bytes) in a column with the LIST OF BYTE VARYING data type. If you omit *n*, SQL creates a 1-octet column. In the chained list format, the maximum size for *n* is 65,508 for the first segment and 65,522 for each subsequent segment. In the indexed list format, the maximum size for *n* is 65,530, leaving 5 bytes for overhead. See Section 2.3.7.1 for more information on the chained and indexed formats for lists.

The user data portion of a list segment is a field of data type BYTE VARYING. The BYTE VARYING data type is a string of unsigned 8-bit bytes. The data type BYTE VARYING, reserved for future use, is currently only valid within a list segment.

You must use the CREATE TABLE statement to create a list because a list is stored within a row in a table. In fact, you store a segmented string identifier in the column with the LIST OF BYTE VARYING data type. The **segmented string identifier** is a number that specifies the location of the primary list segment. In indexed list format, the segmented string identifier points to the first pointer segment. In chained list format, the segmented string identifier points to the first list segment. Because you store a pointer to the list table, rather than the list itself, the list is not constrained by the Oracle Rdb table size limit. For an example of creating a table that contains a list, see the CREATE TABLE Statement.

For more information about using lists, see the DECLARE CURSOR Statement. For information about storing lists in separate storage areas from other table information, see the CREATE STORAGE MAP Statement.

The LIST OF BYTE VARYING data type supports BINARY and TEXT as subtypes. Use these subtypes to specify the data contained within a LIST OF BYTE VARYING data type. The subtype TEXT specifies that the data type can contain any printable characters. The subtype BINARY specifies that the data type contains raw binary data represented in hexadecimal notation.

2.3.7.1 On-Disk Format of Lists

Oracle Rdb provides three on-disk formats for lists:

- Chained format
- Indexed format
- Single-segment format

In the original chained format, lists are a chained list of segments. The first segment contains a pointer to the second segment, the second segment contains a pointer to the third, and so forth. The final segment contains a null pointer.

Each segment contains an 8-byte database key (dbkey) pointing to the next segment, leaving 65,522 bytes for user data.

The first segment includes 14 bytes of overhead to describe the segmented string:

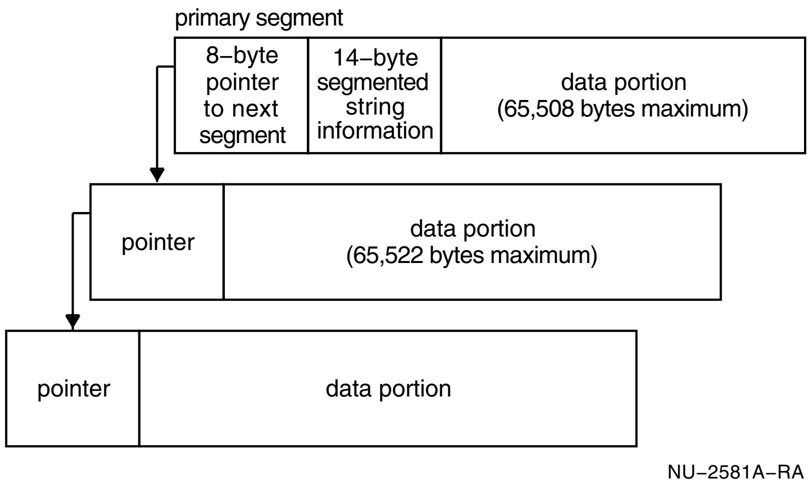
- A quadword that contains the length of the entire string
- A longword that contains the total number of segments

- A word that contains the length of the longest segment

Due to this overhead, the first segment can hold only 65,508 bytes of user data.

This information held in the first segment is returned in the SQLCA structure when SQL is used to open a list cursor. Figure 2–3 shows the chained list format.

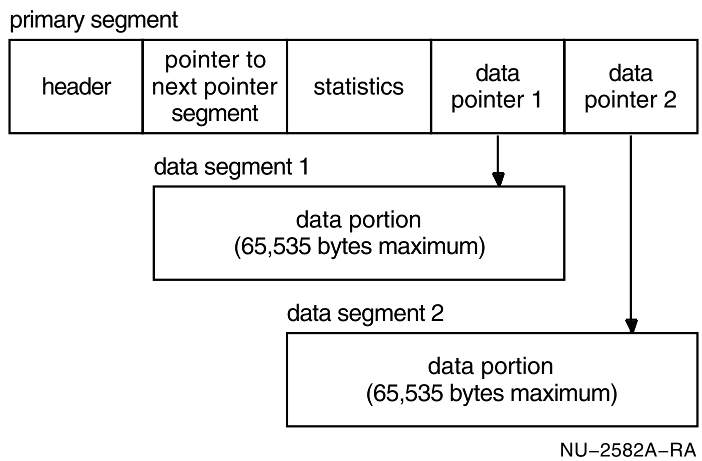
Figure 2–3 Chained List Format



The initial segment in this chained list format contains information that is not available until after all segments are written to the disk. This style inherently requires updating, so an indexed list format was developed to alleviate this problem.

In the indexed list format, data segments no longer contain a pointer to the next segment. Instead, the pointers are kept in special segments called pointer segments. A pointer segment contains only pointers to data segments. Figure 2–4 shows the structure of the indexed list format.

Figure 2–4 Indexed List Format



The pointer segments allow Oracle Rdb to write the data segments without needing to revise them later. A pointer segment is sized according to the free space on a page of the storage area. If there is no more free space on the page to store a data segment, the pointer is chained with a new pointer segment. This format, combined with buffering and large page sizes, virtually eliminates the need to revise pages.

The indexed list format is the default for all lists created by Oracle Rdb.

To retain the chained list format as the default, you must define the logical name RDMS$USE_OLD_SEGMENTED_STRING.

```
$ DEFINE RDMS$USE_OLD_SEGMENTED_STRING YES
```

When this logical name is defined, it causes the application to write the chained list format to all read/write media.

If you want to use the new indexed list format at a later time, you must deassign the RDMS$USE_OLD_SEGMENTED_STRING logical name.

Mixing chained and indexed list formats in the same table is supported. However, you may want to convert your chained list format to the indexed list format. For example, conversion is desirable if you want to perform FETCH LAST statements with a scrollable list cursor. With chained list format, a FETCH LAST statement causes Oracle Rdb to read all segments before reaching the desired segment; this is not optimal. With indexed list format, a FETCH LAST statement causes Oracle Rdb to read only the pointer segment and the last data segment.

To prevent performing a FETCH LAST statement with chained list format, define the logical name RDMS$SET_FLAGS, or use SET FLAGS with 'NOSCROLL_EMULATION.' Defining this logical name causes an OPEN CURSOR statement to fail if it tries to open a SCROLL list cursor on the chained list format.

You can see a demonstration of the conversion process from chained list format to indexed list format in the sample program sql_convert_lists.sc, in the sample directory.

Lists can also be formatted in single segments if the amount of data can fit within the segmented string buffer, which is controlled by the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name.

A single-segment list consists of a field that is used to differentiate it from primary segments and data segments. This helps reduce disk storage by omitting pointers and other overhead. Only a single I/O is necessary to read the segment. To take advantage of single-segment lists, do not define the RDMS$USE_OLD_SEGMENTED_STRING logical name.

2.3.8 Data Type Conversions

Two levels of data type conversion can take place when values are assigned from SQL to a host language parameter or from a host language parameter to SQL.

- Conversion from a data type that is not supported by the database, or conversion from a data type that is not supported by a host language to a data type that is supported

  SQL allows programs to declare host language parameters with certain data types that are not supported by databases underlying SQL. (SQL has no corresponding data type for COBOL COMP-2 or COMP-3 data, for example.) Similarly, SQL supports data types for which some languages do not have a corresponding data type. (PL/I does not support BIGINT data, for example.) In both instances, data is converted from the unsupported source data type to a supported target data type.

  The specific conversions that take place at this level depend on the host language. Section 2.3.8.1 describes this level of conversion.

- Conversion from one data type to another

  SQL generally allows assignment of a value between two different data types it supports (such as CHAR and DATE). This level of conversion is independent of the host language.

Section 2.3.8.2 describes the rules for converting data between supported data types.

Note

Oracle Rdb encourages application programs to use the CAST function to explicitly convert the data to consistent and comparable format to avoid the problems often encountered when integer and text values are compared.

2.3.8.1 Conversion from Unsupported Data Types

Databases and the various host languages supported by the SQL precompiler or module language processor do not necessarily support the same set of data types. SQL handles this incompatibility between databases and the different languages in one of the following ways:

- SQL converts database values to the host language data type, and host language values to the supported data type. SQL makes this conversion only for a subset of valid host language declarations.
- SQL generates an error when it precompiles the program.

Section 4.4 describes which host language declarations SQL converts to and from for languages supported by the precompiler. Section 3.5 contains tables showing host language declarations that are compatible with parameters declared in SQL modules. Such host language parameter declarations must correspond exactly to the corresponding formal parameter declarations in the SQL module file. If they do not, the program can generate unpredictable results at run time. Appendix D describes how SQL converts program and database data types in dynamic SQL.

Note

None of the host languages that work with the SQL precompiler supports the DATE data type. SQL does not convert DATE, TIME, TIMESTAMP, or INTERVAL values to the host language data types shown in Tables 4–3 through 4–10. Instead, SQL assigns the 64-bit value stored in a DATE, TIME, TIMESTAMP, or INTERVAL column to parameters declared, as shown in those tables.

Once the value is stored in the parameter, programs can use the LIB$FORMAT_DATE_TIME Run-Time Library routine to convert the

64-bit value to an ASCII string for the DATE, TIME, TIMESTAMP, and INTERVAL data types.

2.3.8.2 Conversion Between Supported Data Types

In general, SQL allows assignments between supported data types. In such assignments, the underlying database system converts the data type from that of the source column or parameter to that of the target column or parameter.

Conversions between character data types follow these rules:

- If automatic translation has not been enabled then the character sets of the source string and the target string must be identical.
- If the source string is longer than the target string, the result is left-justified and truncated on the right with no error reported for dialects MIA, SQL89, and SQLV40.

  If you use a modern dialect such as SQL99, 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 the truncation splits a multi-octet character from a mixed multi-octet character set, SQL replaces the bytes in the incomplete character in the target string with ASCII space characters.

  If the truncation splits a multi-octet character from a fixed multi-octet character set, SQL replaces the bytes in the incomplete character in the target string with the low-order octet of the appropriate space character for any multi-octet character set.
- If the source string is shorter than the target string, the result is left-justified and filled on the right with the appropriate space character. There is an exception to this rule: If the column in a table is defined as CHAR or CHAR(1) in the C language, the target string is terminated with a null character instead of being filled with blank spaces, generating a string of length 1.
- If a text data item with trailing blank spaces is assigned to a varying string data item, the trailing blanks are considered part of the length of the field.

Conversions between fixed-point numeric data types follow these rules:

- If the source has more fractional places than the target can hold, the result is rounded off.

- If the source uses more integer places than the target can hold, an arithmetic error is returned.
- If rounding off the decimal portion causes the integer portion to overflow the target, an arithmetic error is returned.
- If the target has more integer or decimal places than the source, the result is extended with zeros to the right or left, as appropriate.

Conversions between floating-point numeric data types follow these rules:

- If the source has more precision than the target, the low-order portion of the source is rounded off.
- If the target cannot express the magnitude of the source, an arithmetic error is returned.

Conversions for the LIST OF BYTE VARYING data type are not supported. You can, however, convert an element with the LIST OF BYTE VARYING data type to data type CHAR or VARCHAR if the language you are using supports it.

Conversions between data that have different data types follow these rules:

- Text to be converted to a numeric data type must contain text that represents a number either in decimal format or scientific notation with no commas.

  Numeric data converted to text produces a decimal-format number from fixed-point data and scientific notation format from floating-point data.
- For conversions from numeric data types to an INTERVAL data type, you must use the CAST operator. The output type is restricted to an INTERVAL containing only a single date-time field in the interval qualifier.
- In assignments from the DATE data type to CHAR or VARCHAR, two different output formats are available. This document refers to these formats as VMS format and ANSI format.
  - In assignments from text to DATE VMS, the text expression must contain ASCII digits representing a date in the format dd-mmm-yyyy hh:mm:ss.cc, which is translated in Table 2–12.
  - In assignments from DATE VMS to text, SQL converts to the text format described in Table 2–12. If the text field is less than 16 characters, the output is truncated from the right, losing hundredths of seconds first and the first digit of the year last. For example, the date 1990112523053488 (which can be expressed as the literal '25-Nov-1990 23:05:34.88') would be truncated to 199011252305 if the text field had only 12 characters.

If the text field is longer than 16 characters, the field is left-justified and blank-filled. The text expression appears in the format shown in Table 2–12.

The following example shows this DATE VMS format:

```
SQL> -- Oracle Rdb supports another internal format for
SQL> -- DATE VMS.  This format is similar to the TIMESTAMP format
SQL> -- except that the punctuation is omitted.  This example assigns
SQL> -- the CHAR column (A) to the DATE VMS (G) column and then
SQL> -- displays the result.  In an application, the CHAR column could
SQL> -- be a CHAR host variable or module language parameter.
SQL> --
SQL> INSERT INTO DATE_TEST (G) VALUE ('1957020100000000');
1 row inserted
SQL> UPDATE DATE_TEST
cont>   SET A=G
cont>   WHERE G IS NOT NULL;
1 row updated
SQL> SELECT A, G
cont>   FROM DATE_TEST
cont>   WHERE G IS NOT NULL;
 A                          G
  1-FEB-1957 00:00:00.00   1957020100000000
1 row selected
SQL> ROLLBACK;
```

– In assignments from text to DATE VMS, the text expression must contain ASCII digits representing a date in the format shown in Table 2–12.

  If the input text expression is more than 16 characters, only the first 16 characters are used. The rest of the input is ignored.

  If the input text expression is between 8 and 15 characters, it is treated as though it were filled with ASCII zeros on the right, up to 16 characters.

Table 2–12 Format of Text Strings Converted to or from DATE VMS Data Type

String	Meaning
yyyy	Four digits of year, between 1857 and 9999
mmm	First 3 characters of the month name (for example, JAN)
nn	Two digits of month, including leading zero for months between January and September, between 01 and 12

(continued on next page)

Table 2–12 (Cont.) Format of Text Strings Converted to or from DATE VMS Data Type

String	Meaning
dd	Two digits of day of month, 01 to 31, right-justified and zero-filled
hh	Two digits of hour of day on a 24-hour clock, 00 to 23, right-justified and zero-filled
mm	Two digits of minute of hour, 00 to 59, right-justified and zero-filled
ss	Two digits of second of minute, 00 to 59, right-justified and zero-filled
cc	Two digits of fractions of a second, 00 to 99, right-justified and zero-filled

If the input text expression is less than 8 characters, the assignment returns a conversion error.

- In assignments from text to ANSI format DATE, the text expression must contain ASCII digits representing a date in the following formats:
  - TIME — hh:mm:ss.cc
  - DATE — yyyy-nn-dd
  - TIMESTAMP — yyyy-nn-dd hh:mm:ss.cc
  - INTERVAL (YEAR-MONTH) — y-m
  - INTERVAL (DAY-TIME) — d:hh:mm:ss.cc

- When you use the precompiler, module language, or dynamic SQL, display operations should always use CAST or EXTRACT with CHAR host variables to convert date-time data from binary data.

Table 2–13 shows when data type conversions are allowed between data types and what special conditions can apply to such conversions. Note that:

- Yes: means a conversion is allowed and will be attempted.
- No: means the data types are not compatible.
- N/A: means conversion rules for these data types are already defined.

Table 2–13 Conversion Rules

Source Data Types	Target Data Types: DATE	TIME	TIME-STAMP	ADT	INTERVAL year-month	INTERVAL day-time	Numeric	Text[1]
DATE ANSI	Yes	No	Yes[2]	Yes[2]	No	No	No	Yes[7]
TIME	No	Yes	Yes[3]	Yes[10]	No	No	No	Yes[7]
TIMESTAMP	Yes	Yes[4]	Yes	Yes	No	No	No	Yes[7]
ADT	Yes	Yes[4]	Yes	Yes	No	No	No	Yes[7]
INTERVAL year-month	No	No	No	No	Yes	No	Extract[5]	Yes[7]
INTERVAL day-time	No	No	No	No	No	Yes	Extract[5]	Yes[7]
Numeric	No	No	No	No	Cast[6]	Cast[6]	N/A	Yes[7]
Text	Yes[8]	Yes[8]	Yes[8]	Yes[8]	Yes[8]	Yes[8]	N/A	Yes[9]

[1]Text can be CHAR (TEXT), NCHAR, VARCHAR (VARYING STRING), NCHAR VARYING, or LONG VARCHAR data types. CHAR and VARCHAR can be qualified by the name of a character set.

[2]The TIME portion is 00:00:00.00.

[3]The DATE portion defaults to the CURRENT_DATE.

[4]The DATE portion is discarded.

[5]You must use the EXTRACT built-in function.

[6]You must use the CAST built-in function, and output must be a single-field interval.

[7]The target character set must contain ASCII. SQL converts the value to an appropriate ASCII representation.

[8]The source character set must contain ASCII, and the value must be presented in ASCII.

[9]If automatic translation has not been enabled then the character sets must be identical.

[10]The DATE portion defaults to 17-NOV-1858.

Valid assignments include:

- TEXT can be assigned to TIMESTAMP, TIME, DATE, and INTERVAL. The syntax must conform to that defined in Section 2.4.3.
- If TIMESTAMP, TIME, DATE (ANSI), or INTERVAL is directly assigned to CHAR or VARCHAR, then the output will be in the ANSI literal format.
- If DATE (VMS) is directly assigned to CHAR or VARCHAR, then the output will be the format shown in Table 2–12.

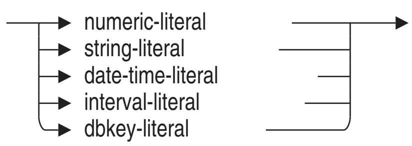
- Numeric data types can be converted to an INTERVAL data type only using the CAST operator. The output is restricted to an INTERVAL data type containing only a single date-time field in the interval qualifier. SQL allows only one field in the interval qualifier.

2.4 Literals

Literals, which are also called constants, specify a value.

The following diagram shows the format of literals:

literal =



Literals are a type of value expression (see Section 2.6). Many SQL clauses that do not accept general value expressions require literal values. Literal values can be either numeric, character string, or date. In addition, SQL provides keywords that refer to literals, for example:

- NULL
- CURRENT_DATE
- SYSTEM_USER

The following sections describe each type of literal.

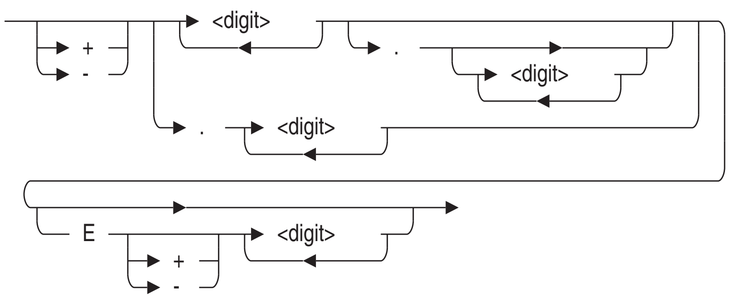
2.4.1 Numeric Literals

A numeric literal is a string of digits that SQL interprets as a decimal number. A numeric literal can be a:

- Decimal string that consists of digits and an optional decimal point. The maximum length, not counting the decimal point, is 19 digits.
- Decimal number in scientific notation (E notation) that consists of a decimal string mantissa and a signed integer exponent, separated by the letter E. You cannot embed spaces in E notation.

The following syntax shows the format of numeric literals:

numeric-literal =



SQL allows flexibility in numeric expressions. You can use unary plus and minus, and you can use any form of decimal notation. The following are valid numeric strings:

123
34.9
–123
.25
123.
0.33889909
6.03E+23
6.03E–23

If you use a numeric literal to assign a value to a column or a parameter, the data type of the column or parameter determines the maximum value you can assign and whether or not you can assign values to the right of the decimal point. If the data type of the column or parameter is different from the implied data type of the numeric literal, SQL converts the literal to the data type of the column or parameter.

Section 2.3 specifies the range of values allowed for a numeric literal assigned to each SQL data type.

2.4.2 Character String Literals

SQL recognizes the following types of character string literals:

- A quoted character string to represent printable characters from the session's literal character set.
- A quoted character string qualified by the name of a character set. The string represents printable characters from the named character set.

- A **national character string literal** (an *N* followed by a quoted character string), represents printable characters from the national character set.
- A hexadecimal character string (an *X* followed by a quoted character string) represents printable and nonprintable ASCII characters.

Section 2.4.2.1 and Section 2.4.2.2 describe both types of character string literals.

2.4.2.1 Quoted Character String Literals

A quoted character string literal is a string of printable characters enclosed in single quotation marks. The maximum length of a character string is 1,024 octets. An unqualified character string must contain characters only from the literal character set of that session.

The printable ASCII characters consist of:

- Uppercase alphabetic characters:

  A–Z
- Lowercase alphabetic characters:

  a–z
- Numerals:

  0–9
- Special characters:

  ```
  ! @ # $ % ^ & * ( ) - _ = + ` ~
  [ ] { } ; : " \ | / ? > < . ,
  ```

For a list of the printable characters for DEC_MCS, see the OpenVMS documentation for users; for a list of printable characters for the other supported character sets, see the standard for that character set. Section 2.1 lists the standards for each character set.

Use a pair of single quotation marks to enclose a character string literal. If you use double quotation marks, an informational message is displayed, indicating that double quotation marks are nonstandard. Double quotation marks are passed as delimited identifiers if the quoting rules are set to ANSI/ISO SQL. See the SET QUOTING RULES Statement for information on setting quoting rules. When using quotation marks, follow these rules:

- Begin and end a character string literal with the same type of quotation mark.
- To include double quotation marks in a character string literal, enclose the character string in single quotation marks.

- If a quotation mark appears in a character string literal enclosed by quotation marks, use two consecutive quotation marks for every one you want to include in the literal. This technique is necessary if you want to include quotation marks of both types in one quoted string. See Table 2–14 for examples using quotation marks.
- Ensure that the contents of the quoted string contain an integral number of characters equal to the minimum number of octets needed for the specified character set. For example, a Kanji character requires a minimum of 2 octets (or 2 bytes). Therefore, the quoted string must contain a total number of octets that is a multiple of 2. If you try to insert a quoted string that contains 3 octets, SQL interprets the ending single quotation (') mark as the 4th octet instead of the string terminator and returns an error, as shown in the following example:

```
SQL> INSERT INTO COLOURS
cont> (JAPANESE)
cont> VALUES
cont> (N'黒2');
%SQL-F-UNTSTR, Unterminated string found
SQL> !
SQL> ! SQL returns an error because the character 2 is a
SQL> ! one-byte ASCII character and the national character
SQL> ! set is KANJI which requires a two-byte character.
SQL> ! The next command uses the two-byte version of the
SQL> ! character 2.
SQL> !
SQL> INSERT INTO COLOURS
cont> (JAPANESE)
cont> VALUES
cont> (N'黒２');
1 row inserted
```

Table 2–14 shows how to use quotation marks in character string literals.

Table 2–14 Embedding Quotation Marks in Literals

This String:	Is Interpreted As:
'UNQUOTED LITERAL'	UNQUOTED LITERAL
'"A LITERAL WITH QUOTES"'	"A LITERAL WITH QUOTES"
'''ANOTHER ONE'''	'ANOTHER ONE'
'RICHARD "RICK"SMITH''S'	RICHARD "RICK"SMITH'S

(continued on next page)

Table 2–14 (Cont.) Embedding Quotation Marks in Literals

This String:	Is Interpreted As:
'Richard ''Rick''Smith ''s'	Richard 'Rick'Smith's
'''' '''''' '	''''
'"'	"
"JONES'	[invalid]

Note

SQL preserves the case distinction in character string literals. That is, NAME = 'JONES' and NAME = 'Jones' yield different results. See Section 2.7 for more information about comparisons.

2.4.2.1.1 Quoted Character String Literals Qualified by a Character Set

You can use a quoted character string literal qualified by the name of a character set. The character string must contain characters only from the named character set.

A string literal qualified by a character set begins with an underscore (_), followed by the name of a supported character set, and a quoted string. No blank spaces are allowed outside of the literal.

The following example shows how to qualify character strings with DEC_MCS and with DEC_KANJI:

```
_DEC_MCS'Blue'
```

```
_DEC_KANJI'Blue'
```

See Section 2.1 for the names of supported character sets.

2.4.2.1.2 Quoted Character String Literals Qualified by the National Character Set

You can use a national character string literal, which is a quoted character string literal qualified by the national character set. The character string must contain characters only from the national character set.

A national character string literal begins with the letter *N* followed by a quoted string. No blank spaces are allowed outside of the literal.

The following example shows how to qualify a character string with the national character set:

```
N'Blue'
```

See Section 2.1.7 for information about the national character set.

2.4.2.2 Hexadecimal Character String Literals

A hexadecimal character string literal begins with an *X* followed by a string of up to 16 characters enclosed in single quotation marks. This type of string literal lets you represent nonprintable ASCII characters by specifying the hexadecimal value of the characters within the quotation marks.

Each ASCII character requires 2 hexadecimal digits to represent it, so you must provide an even number of characters within the quotation marks. The only valid characters for hexadecimal character string literals are 0 through 9 and *A* through *F* (uppercase or lowercase).

In the following example, the hexadecimal character string literal represents two delete characters; the ASCII hexadecimal value for a delete character is *FF*:

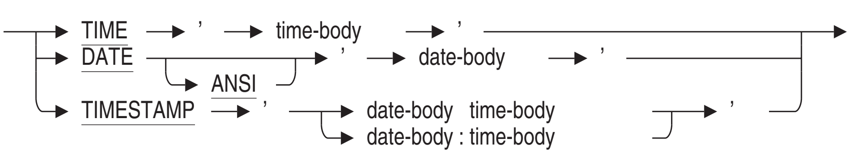
```
X'FFFF'
```

2.4.3 Date-Time Literals

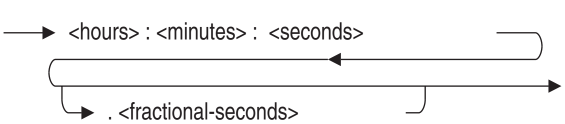
When you refer to a date-time data type with a literal in an SQL statement, you must precede the literal with the data type name and enclose the literal in single quotation marks. You must provide values for all fields, and values must be within the valid range for the field.

The following syntax shows the format of date-time literals:

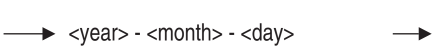
date-time-literal =



time-body =



date-body =



Note

In the following syntax descriptions, *y*, *d*, *h*, and *s* stand for single digits in fields representing years, days, hours, and seconds, respectively. The letter *m* stands for 1 digit of the month number when it follows a *y*, and 1 digit of the minutes number when it does not. Fractions of a second are represented by digits after the decimal point.

The syntax for date-time literals is as follows:

- DATE literals

  DATE 'yyyy-mm-dd'
  or
  DATE 'dd-mmm-yyyy hh:mm:ss.ss'

  ```
  Examples: DATE ANSI '1993-05-27'
            DATE VMS '27-MAY-1993 15:25:00.00'
  ```

  SQL includes leap year validation for the 29th of February.

- TIME literals

  TIME 'h:m:s'
  TIME 'h:m:s.s'

  ```
  Example: TIME '14:23:45.19'
  ```

  TIME represents 24-hour time.

- TIMESTAMP literals

  TIMESTAMP 'y-m-d h:m:s'
  or
  TIMESTAMP 'y-m-d:h:m:s'

  ```
  Example: TIMESTAMP '1993-1-4 14:12:01.00'
           TIMESTAMP '1993-1-4:14:12:01.00'
  ```

  There are two formats allowed for the TIMESTAMP literal. The SQL92 format allows a separating space character between the date-body and the time-body as shown in the previous example. The nonstandard format allows a separating colon character between the date-body and the time-body. For example:

```
SQL> SET DEFAULT DATE FORMAT 'SQL92';
SQL> --
SQL> -- Create a table and insert several rows using the SQL92 format and
SQL> -- the nonstandard format for the TIMESTAMP literal.
SQL> --
SQL> CREATE TABLE t (a INTEGER, b TIMESTAMP(2)
cont>                           DEFAULT TIMESTAMP '1995-1-1 12:34:10.01');
SQL> INSERT INTO t (a) VALUE (0);
1 row inserted
SQL> --
SQL> -- Insert a row using the nonstandard format for the TIMESTAMP
SQL> -- literal.
SQL> --
SQL> INSERT INTO t (a,b) VALUE (1, TIMESTAMP '1995-1-1:12:34:10.01');
1 row inserted
SQL> --
SQL> -- Insert a row using the SQL92 format for the TIMESTAMP literal.
SQL> --
SQL> INSERT INTO t (a,b) VALUE (2, TIMESTAMP '1995-1-1 12:34:10.01');
1 row inserted
SQL> --
SQL> -- Select the rows.  SQL uses the SQL92 format to display the
SQL> -- TIMESTAMP literal for all selected rows.
SQL> --
SQL> SELECT a, b, CAST (b AS CHAR(30)) FROM t ORDER BY a;
          A   B
          0   1995-01-01 12:34:10.01   1995-01-01 12:34:10.01
          1   1995-01-01 12:34:10.01   1995-01-01 12:34:10.01
          2   1995-01-01 12:34:10.01   1995-01-01 12:34:10.01
3 rows selected
```

- INTERVAL literals

  INTERVAL '±y-m' YEAR TO MONTH
  INTERVAL '±d:h:m:s.s' DAY TO SECOND

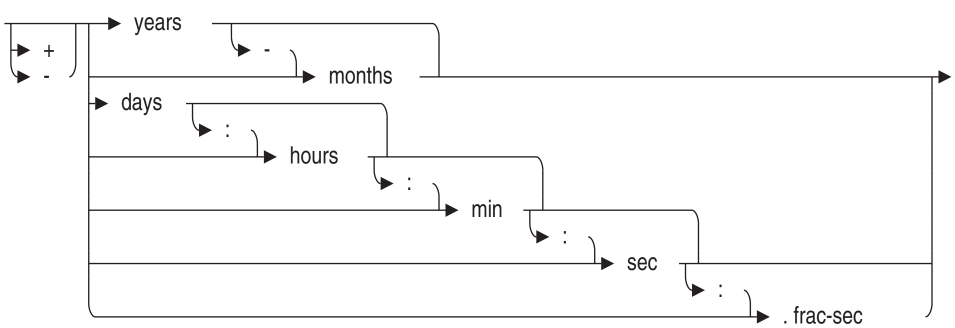
```
Examples: INTERVAL '-1-2' YEAR TO MONTH
          INTERVAL '1:4:30:0.0' DAY TO SECOND
          INTERVAL '1:10' DAY TO HOUR
          INTERVAL '235' MONTH(3)
```

  The following syntax shows the format of interval literals:

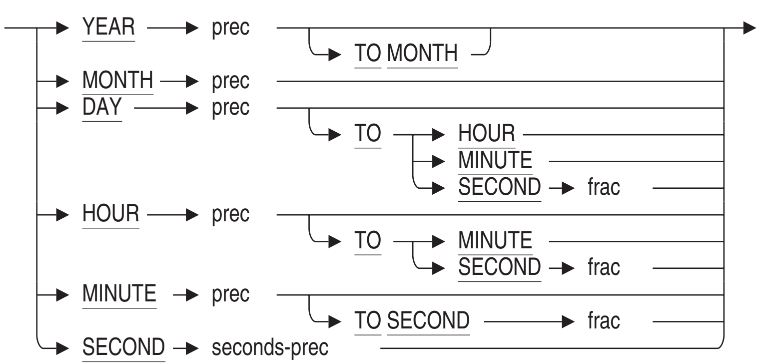
```
interval-literal =

──► INTERVAL ──► ' ──► <interval-body> ──► ' ──► interval-qualifier ──►
```

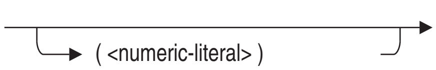
interval-body =



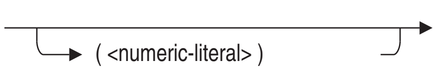
interval-qualifier =



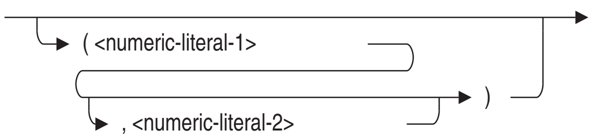
frac =



prec =



seconds-prec =



Because intervals can be signed quantities, a leading addition or subtraction operator can precede the literal to indicate positive (+) or negative (–) intervals.

You must specify an appropriate interval qualifier in each interval literal. The INSERT statement in the following example specifies an interval qualifier that is too small:

```
SQL> -- Create a table with a field of interval month(4).
SQL> CREATE TABLE TEST_TABLE (TEST_COL INTERVAL MONTH(4));
SQL> --
SQL> -- Insert into the field using the literal INTERVAL '200' MONTH.
SQL> INSERT INTO TEST_TABLE (TEST_COL) VALUE (INTERVAL '200' MONTH);
%SQL-F-DATCONERR, Data conversion error for string '200'
-COSI-F-IVTIME, invalid date or time
SQL> --
SQL> -- The INTERVAL literal used does not provide a large enough
SQL> -- leading-field precision. The default leading-field precision is 2,
SQL> -- and 200 requires a minimum of 3 because it is 3 digits.
SQL> -- To avoid the error, specify 3 as the leading-field precision
SQL> -- instead of relying on the default.
SQL> INSERT INTO TEST_TABLE (TEST_COL) VALUE (INTERVAL '200' MONTH(3));
1 row inserted
```

In addition to these default formats, you can specify alternate formats for the output display of time and date values using the SET DATE FORMAT statement. (These alternate formats affect only date string text literals and their conversion to and from binary dates. Dates supplied by host languages in 8-byte (64-bit) OpenVMS date and time are not affected by the SET DATE FORMAT statement.)

You can use the SET DATE FORMAT statement only to format columns with the DATE VMS data type. The SET DATE FORMAT statement changes only the output for the date or time formats or both. If you want to change the input format, use the logical name LIB$DT_INPUT_FORMAT. See the OpenVMS run-time library documentation for more information about the LIB$DT_INPUT_FORMAT logical name.

See the SET Statement for complete information on the SET DATE FORMAT statement.

Note

Three character string literals that are translated into DATE VMS format (the dd-mmm-yyyy 00:00:00.00 format explained in Section 2.4.3) are YESTERDAY, TODAY, and TOMORROW. This translation takes place at compile time. In interactive SQL, the dates into which the YESTERDAY, TODAY, and TOMORROW literals are translated are relative to the day when the statement containing the literals is executed.

However, when a program containing the YESTERDAY, TODAY, and TOMORROW literals is processed by the precompiler or in SQL module language, the dates into which the literals are translated at run time are relative to the compile time. In other words, if you compile a program containing these literals on January 4, 2003, YESTERDAY translates to 03-JAN-2003, TODAY translates to 04-JAN-2003, and TOMORROW translates to 05-JAN-2003, regardless of the day the program is run. Oracle does not recommend this method.

Oracle recommends using the following:

- CURRENT_DATE - INTERVAL '1' DAY for YESTERDAY
- CURRENT_DATE for TODAY
- CURRENT_DATE + INTERVAL '1' DAY for TOMORROW
- Use CAST (. . . AS DATE VMS) as required

- DBKEY string literals

  THE DBKEY literal is used primarily by database administrators who have database keys which were displayed in error messages, or shown on an RMU/SHOW STATISTICS display and wish to display the associated row.

  The dbkey string literal is prefixed by _DBKEY, or _ROWID to identify it as a special DBKEY literal. Some examples of valid DBKEY literals are as follows:

  - _DBKEY'23:5628:0'

    An Oracle Rdb table dbkey has three parts, a logical area (in this example 23), a page number (in this example 5628), and a line number (in this example 0). All three parts must be specified.

- _ROWID'23:5628:0, 45:345:15'

  The DBKEY string literal may include several comma separated dbkeys if this is used to reference a view table. Each DBKEY references a row from the view made up of component rows from a table.

  The ROWID keyword is a synonym for DBKEY.

Leading and trailing spaces are ignored, however, spaces may not be embedded within the numeric values in the DBKEY.

Errors will be reported if the DBKEY is for a different table, is incorrectly formatted, or does not reference a row. The reported errors are shown in the following example. A question mark is placed within the string to highlight the syntax error.

```
SQL> select * from employees where dbkey = _dbkey'1,2,3';
%RDB-F-CONVERT_ERROR, invalid or unsupported data conversion
-RDMS-E-DBKFORMAT, database key format incorrect "1,?2,3" - unexpected
character
SQL> select * from employees where dbkey = _dbkey'-1:+2:0';
%RDB-F-CONVERT_ERROR, invalid or unsupported data conversion
-RDMS-E-DBKFORMAT, database key format incorrect "-1:+?2:0" - unexpected
character
SQL> select * from employees where dbkey = _dbkey'23:1:1';
%RDB-E-NO_RECORD, access by dbkey failed because dbkey is no longer associated
with a record
-RDMS-F-INVDBK, 23:1:1 is not a valid dbkey
```

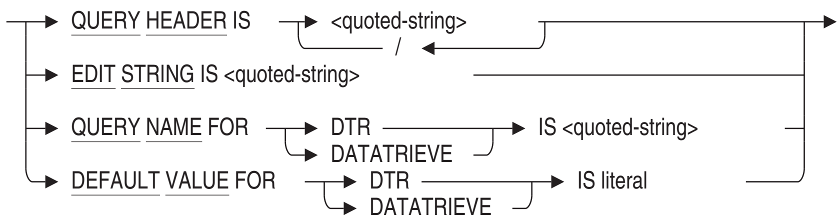
2.5 SQL and DATATRIEVE Formatting Clauses

Optional SQL and DATATRIEVE formatting clauses allow you to modify data displays or query characteristics for interactive SQL users, DATATRIEVE users, or both. The optional formatting clauses (QUERY NAME and EDIT STRING) and DATATRIEVE clauses (QUERY HEADER and DEFAULT VALUE) can be used with the following statements:

- CREATE TABLE
- CREATE DOMAIN
- CREATE VIEW
- ALTER TABLE
- ALTER DOMAIN

The following diagram shows the format for these clauses:

sql-and-dtr-clause =



- A **query header** specifies a string, enclosed in quotation marks, that interactive SQL or DATATRIEVE displays in place of the column name when it retrieves values from a column. Query headers allow you to specify descriptive headings for columns.

  Both interactive SQL and DATATRIEVE display any query headers you specify in SQL definitions.

- An **edit string** specifies a string, enclosed in quotation marks, that controls how interactive SQL or DATATRIEVE formats the display of values in a column.

  Both interactive SQL and DATATRIEVE use edit strings you specify in SQL definitions to control display formatting for those definitions.

  DATATRIEVE recognizes columns with null values and displays them according to the edit string for the missing value.

- A **query name** specifies a string, enclosed in quotation marks, that you can use instead of the column name when formulating DATATRIEVE queries. Query names are useful for abbreviating long column names in DATATRIEVE queries.

  SQL does not recognize query names in interactive queries; the QUERY NAME clause is useful only when you use DATATRIEVE to retrieve the data.

- If you specify a **default value** for a column and do not specify that column in a DATATRIEVE STORE or MODIFY statement, DATATRIEVE stores the default value specified in the SQL definition.

  SQL does not recognize default values in INSERT or UPDATE statements; the DEFAULT VALUE clause is useful only when you use DATATRIEVE STORE or MODIFY statements.

See the DATATRIEVE documentation for additional details.

The following sections describe the SQL formatting clauses, QUERY HEADER and EDIT STRING, in detail.

2.5.1 QUERY HEADER Clause

The QUERY HEADER clause specifies the column header that SQL uses in displays of result tables that contain that column.

If you include the QUERY HEADER clause, SQL uses the query header as the column header. If you omit the clause, SQL uses the column name as the column header.

The column header can include any character except a carriage return, a line feed, or a control character. To include a double quotation mark in a column header, enclose it in single quotation marks.

The following example defines a query header for one column and a DATATRIEVE query name for another column:

```
SQL> ALTER TABLE TEMP
cont>      ADD STATE CHAR (2)
cont>         QUERY NAME FOR DATATRIEVE IS 'ST'
cont>      ADD SEX CHAR (1)
cont>         QUERY HEADER IS 'S'/'E'/'X';
```

These statements define query headers and query names for the STATE and SEX columns. The slash character (/) specifies that the header is split into three lines, so the header for the SEX column is 1 character wide, like the column itself.

Both SQL and DATATRIEVE display the query header used in this example. Only DATATRIEVE recognizes the query name.

2.5.2 EDIT STRING Clause

The EDIT STRING clause specifies the output format of a column value. SQL uses the EDIT STRING clause as the default format when writing a column value to a file or output device.

To specify the format of a column value, use a string of one or more edit characters. Specify the edit string characters in single quotation marks without embedded spaces. In general, each edit character corresponds to 1 character position in the displayed output. For example, 999999 specifies that the output is 6 digits in 6 character positions.

To enter more of the same edit characters, shorten the edit string by placing a repeat count in parentheses following the edit character. For example, the edit string 9(6) is equal to 999999.

You can change the character that SQL and DATATRIEVE display for the currency symbol ($), decimal point (.), and digit separator (,) edit string characters.

To make your output conform to other conventions for numeric and monetary notation, override the system defaults for these symbols by redefining the following logical names:

- SYS$CURRENCY: Specifies the character SQL substitutes for the dollar sign ($) edit string character. The default is a dollar sign.
- SYS$RADIX_POINT: Specifies the character SQL substitutes for the decimal point (.) edit string character. The default is a decimal point.
- SYS$DIGIT_SEP: Specifies the character SQL substitutes for the comma (,) edit string character. The default is a comma.

You can also use the SET statement to override these logical names. See the SET Statement for more information.

Tables 2–16 through 2–23 list the edit string characters. When you specify an edit character, you must consider the type of the field: alphabetic, alphanumeric, numeric, or date. Using edit string characters designated as only alphabetic or alphanumeric on numeric fields or vice versa produces unexpected results.

Table 2–15 lists the CDO edit string characters accepted by SQL.

Table 2–15 CDO Edit Strings Supported by SQL

Character Type	CDO Character or String
Alphabetic	A
Alphanumeric	T
	X
Colon	:
Comma	,
Date, Day, and Time	D
	H
	J
	M

(continued on next page)

Table 2–15 (Cont.) CDO Edit Strings Supported by SQL

Character Type	CDO Character or String
	N
	P
	R
	Q
	W
	Y
	%
	*
Decimal point	.
Digit	9
Encoded sign	C
Exponent	E
Floating	S
	Z"string"
	-
	+
	$
Literal	'string'
Blank	B
Minus parentheses	(( ))
Missing separator	?
Repeat count	x(n)

Table 2–16 lists the alphabetic and alphanumeric replacement edit string characters.

Table 2–16 Alphabetic and Alphanumeric Replacement Edit String Characters

<table>
<tr><th>Character Type</th><th>Edit String Character</th><th>Description</th></tr>
<tr><td>Alphabetic Replacement</td><td>A</td><td>Replaces each A with an alphabetic character from the column’s content. Places an asterisk (*) in the position of each digit or nonalphabetic character in the column’s content.
<pre>SQL> ALTER TABLE EMPLOYEES ALTER ADDRESS_DATA_1
cont> EDIT STRING 'A(25)';
SQL> SELECT ADDRESS_DATA_1 FROM EMPLOYEES LIMIT TO 2 ROWS;
 ADDRESS_DATA_1
 *** Parnell Place
 *** Tenby Dr*
2 rows selected</pre></td></tr>
<tr><td>Alphanumeric Replacement</td><td>X</td><td>Replaces each X with one character from the column’s content.
<pre>SQL> ALTER TABLE EMPLOYEES ALTER MIDDLE_INITIAL
cont>   EDIT STRING 'x.';
SQL> SELECT MIDDLE_INITIAL FROM EMPLOYEES LIMIT TO 3 ROWS;
 MIDDLE_INITIAL
 A.
 D.
 NULL
3 rows selected</pre></td></tr>
<tr><td></td><td>T</td><td>Reserves the number of display columns specified for the column text. T edit strings are useful for controlling how long character strings wrap in displays. Edit strings containing a T cannot contain other characters.
<pre>SQL> ALTER TABLE EMPLOYEES ALTER ADDRESS_DATA_1
cont>   EDIT_STRING 'T(5)';
SQL> SELECT ADDRESS_DATA_1 FROM EMPLOYEES;
 ADDRESS_DATA_1
 146
 Parne
 ll
 Place</pre></td></tr>
</table>

Table 2–17 lists the numeric replacement edit string characters.

Table 2–17 Numeric Replacement Edit String Characters

<table>
<tr><th>Edit String Character</th><th>Description</th></tr>
<tr><td>9</td><td>Replaces each 9 with 1 digit from the column's content. Nondigit characters are ignored; the digits are right-justified in the output, and the leading character positions (if any) are filled with zeros.
<pre>SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT
cont>   EDIT STRING '999999999';
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;
 SALARY_AMOUNT
 000026291
 000051712
2 rows selected</pre></td></tr>
<tr><td>Z</td><td>Replaces each Z with 1 digit from the column's content, except for leading zeros in the column's content, which are replaced with blank spaces.
<pre>SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT
cont>   EDIT STRING 'ZZZZZZZZZ';
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;
 SALARY_AMOUNT
     26291
     51712
2 rows selected</pre></td></tr>
<tr><td>*</td><td>Replaces each asterisk (*) with 1 digit from the column's content, except for leading zeros, which are replaced with asterisks.
<pre>SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT
cont>   EDIT STRING '*********';
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;
 SALARY_AMOUNT
 ****26291
 ****51712
2 rows selected</pre></td></tr>
<tr><td>.</td><td>A period (.) specifies the character position of the decimal point.
<pre>SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT
cont>   EDIT STRING 'ZZZZZZ.ZZ';
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;
  26291.00
  51712.00
2 rows selected</pre></td></tr>
</table>

Table 2–18 lists the alphanumeric insertion edit string characters.

Table 2–18 Alphanumeric Insertion Edit String Characters

<table>
<tr><th>Edit String Character</th><th>Description</th></tr>
<tr><td>+</td><td>If only one plus sign (+) is specified for an alphanumeric column, inserts the plus sign (+) in that position.
<pre>
SQL> ALTER TABLE EMPLOYEES ALTER EMPLOYEE_ID
cont>   EDIT STRING ’XX+XXX’;
SQL> SELECT EMPLOYEE_ID FROM EMPLOYEES LIMIT TO 2 ROWS;
 EMPLOYEE_ID
 00+164
 00+165
2 rows selected
</pre></td></tr>
<tr><td>-</td><td>Inserts a hyphen (-) in that character position.
<pre>
SQL> ALTER TABLE EMPLOYEES ALTER EMPLOYEE_ID
cont>   EDIT STRING ’XX-XXX’;
SQL> SELECT EMPLOYEE_ID FROM EMPLOYEES LIMIT TO 2 ROWS;
 EMPLOYEE_ID
 00-164
 00-165
2 rows selected
</pre></td></tr>
<tr><td>.</td><td>Inserts a period (.) in that character position.
<pre>
SQL> ALTER TABLE EMPLOYEES ALTER MIDDLE_INITIAL
cont> EDIT STRING ’X.?’’No middle initial’;
SQL> SELECT MIDDLE_INITIAL FROM EMPLOYEES LIMIT TO 10 ROWS;
 MIDDLE_INITIAL
 D.
 G.
 P.
 O.
 M.
 No middle initial
 I.
 No middle initial
 A.
 E.
10 rows selected
</pre></td></tr>
</table>

(continued on next page)

Table 2–18 (Cont.) Alphanumeric Insertion Edit String Characters

<table>
<tr><th>Edit String Character</th><th>Description</th></tr>
<tr><td>,</td><td>Inserts a comma (,) in that character position.<pre>
SQL> ALTER TABLE EMPLOYEES ALTER EMPLOYEE_ID
cont>   EDIT STRING 'XX,XXX';
SQL> SELECT EMPLOYEE_ID FROM EMPLOYEES LIMIT TO 2 ROWS;
 EMPLOYEE_ID
 00,164
 00,165
2 rows selected
</pre></td></tr>
</table>

Table 2–19 lists the numeric insertion edit string characters.

Table 2–19 Numeric Insertion Edit String Characters

<table>
<tr><th>Edit String Character</th><th>Description</th></tr>
<tr><td>+</td><td>If only one plus sign (+) is specified, places a plus sign (+) if the column's content is positive or places a minus sign (–) if it is negative, in the leftmost character position.<pre>
SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:
SQL> ALTER TABLE TEMP ALTER COL1
cont>   EDIT STRING '+9(9).99';
SQL> SELECT COL1 FROM TEMP;
          COL1
 +000053333.00
 -000053333.00
</pre></td></tr>
<tr><td>–</td><td>If only one minus sign (–) is specified, places a blank space if the column's content is positive or places a minus sign (–) if it is negative, in the leftmost character position.<pre>
SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:
SQL> ALTER TABLE TEMP ALTER COL1
cont>   EDIT STRING '-9(9).99';
SQL> SELECT COL1 FROM TEMP;
          COL1
  000053333.00
 -000053333.00
</pre></td></tr>
</table>

(continued on next page)

Table 2–19 (Cont.) Numeric Insertion Edit String Characters

Edit String Character	Description

.	Inserts the character specified by the logical name SYS$RADIX_POINT (default is a decimal point (.)) in that character position. Put only one decimal point (.) in a numeric edit string. (SYS$RADIX_POINT is supported only on OpenVMS.)
	`SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT`
	`cont>    EDIT STRING 'ZZZZZZ.ZZZZ';`
	`SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;`
	` SALARY_AMOUNT`
	`  26291.0000`
	`  51712.0000`
	`2 rows selected`

,	If all the digits to the left of the comma are suppressed zeros, replaces the comma (,) with a blank space. If not, inserts the character specified by the logical name SYS$DIGIT_SEP (default is a comma) in that character position. (SYS$DIGIT_SEP is supported only on OpenVMS.)
	`SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT`
	`cont>   EDIT STRING 'ZZZ,ZZZ.ZZZZ';`
	`SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;`
	` SALARY_AMOUNT`
	`  26,291.0000`
	`  51,712.0000`
	`2 rows selected`

CR	If the column's content is negative, inserts the letters *CR*. If the column's content is positive, inserts two blank spaces. Put only one *CR* in an edit string, either at the far right or the far left.
	`SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:`
	`SQL> ALTER TABLE TEMP ALTER COL1`
	`cont>        EDIT STRING 'ZZZZZ.ZZCR';`
	`SQL> SELECT COL1 FROM TEMP;`
	`       COL1`
	` 53333.00`
	` 53333.00CR`

(continued on next page)

Table 2–19 (Cont.) Numeric Insertion Edit String Characters

Edit String Character	Description

DB	If the column's content is negative, inserts the letters *DB*. If the column's content is positive, inserts two blank spaces. Put only one *DB* in an edit string, either at the far right or the far left.
	`SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:`
	`SQL> ALTER TABLE TEMP ALTER COL1`
	`cont>        EDIT STRING 'ZZZZZ.ZZDB';`
	`SQL> SELECT COL1 FROM TEMP;`
	`       COL1`
	` 53333.00`
	` 53333.00DB`

(( ))	If the column's content is negative, enclosing an edit string in double sets of parentheses inserts single left and right parentheses before and after the column value.
	`SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:`
	`SQL> ALTER TABLE TEMP ALTER COL1`
	`cont>      EDIT STRING '((9(6).99))';`
	`SQL> -- Equivalent notation: '((999999.99))'`
	`SQL> SELECT COL1 FROM TEMP;`
	`        COL1`
	`  053333.00`
	` (053333.00)`

Table 2–20 lists the alphanumeric and numeric insertion edit string characters.

Table 2–20 Alphanumeric and Numeric Insertion Edit String Characters

Edit String Character	Description

B	Inserts a blank space in that character position.
	
	`SQL> ALTER TABLE EMPLOYEES ALTER EMPLOYEE_ID`
	`cont>   EDIT STRING 'XXXXBX';`
	`SQL> SELECT EMPLOYEE_ID FROM EMPLOYEES LIMIT TO 2 ROWS;`
	` EMPLOYEE_ID`
	` 0016 4`
	` 0016 5`
	`2 rows selected`

0	Inserts a zero in that character position.
	
	`SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT`
	`cont>   EDIT STRING '99999.000';`
	`SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY LIMIT TO 2 ROWS;`
	` SALARY_AMOUNT`
	` 26291.000`
	` 51712.000`
	`2 rows selected`

$	If only one dollar sign ($) is specified, inserts the character specified by the logical name SYS$CURRENCY (default is a dollar sign) in that character position. (SYS$CURRENCY is supported only on OpenVMS.)
	
	`SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT`
	`cont>   EDIT STRING '$9(9)';`
	`SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;`
	` SALARY_AMOUNT`
	` $000026291`
	`        .`
	`        .`
	`        .`
	` $000007089`
	`        .`
	`        .`
	`        .`

(continued on next page)

Table 2–20 (Cont.) Alphanumeric and Numeric Insertion Edit String Characters

<table>
<tr><th>Edit String Character</th><th>Description</th></tr>
<tr><td>%</td><td>Inserts a percent sign (%) in that character position.
<pre>
SQL> CREATE VIEW TEST (SALARY EDIT STRING '$99999.99',
cont>   POINTLESS_PERCENT EDIT STRING '%99.999')
cont>   AS SELECT SALARY_AMOUNT,
cont>   SALARY_AMOUNT/SUM(SALARY_AMOUNT)
cont>   FROM SALARY_HISTORY WHERE SALARY_END IS NULL
cont>   GROUP BY SALARY_AMOUNT;
SQL> SELECT * FROM TEST LIMIT TO 2 ROWS;
    SALARY   POINTLESS_PERCENT
 $08687.00   %01.000
 $08951.00   %01.000
2 rows selected
</pre></td></tr>
<tr><td>/</td><td>Inserts a slash (/) in that character position.
<pre>
SQL> ALTER TABLE EMPLOYEES ALTER LAST_NAME
cont>   EDIT STRING 'XXX/';
SQL> SELECT LAST_NAME FROM EMPLOYEES LIMIT TO 2 ROWS;
 LAST_NAME
 Ame/
 And/
2 rows selected
</pre></td></tr>
<tr><td>Literal</td><td>Inserts the character string literal enclosed in quotation marks in that position. The quotation marks are not inserted in the output.
<pre>
SQL> ALTER TABLE EMPLOYEES ALTER LAST_NAME
cont>   EDIT STRING 'XXX/''Truncated last name';
SQL> SELECT LAST_NAME FROM EMPLOYEES LIMIT TO 2 ROWS;
 LAST_NAME
 Ame/Truncated last name
 And/Truncated last name
2 rows selected
</pre></td></tr>
</table>

Table 2–21 lists the numeric floating insertion edit string characters.

Table 2–21 Numeric Floating Insertion Edit String Characters

<table>
<tr><th>Edit String Character</th><th>Description</th></tr>
<tr><td>$</td><td>If more than one dollar sign ($) is specified to the left of the other edit string characters, suppresses leading zeros and inserts the character specified by the SYS$CURRENCY logical name (default is a dollar sign) to the left of the leftmost digit. (SYS$CURRENCY is supported only on OpenVMS.)
<pre>
SQL> -- Compare this with single $ edit string character:
SQL> ALTER TABLE SALARY_HISTORY ALTER SALARY_AMOUNT
cont>   EDIT STRING '$(9).99';
SQL> SELECT SALARY_AMOUNT FROM SALARY_HISTORY;
 SALARY_AMOUNT
    $26291.00
         .
         .
         .
     $7089.00
         .
         .
         .
</pre></td></tr>
<tr><td>+</td><td>If more than one plus sign (+) is specified to the left of the other edit string characters, suppresses leading zeros and displays the sign of the column's value (plus or minus) to the left of the leftmost digit.
<pre>
SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:
SQL> -- Compare this with single + edit string character:
SQL> ALTER TABLE TEMP ALTER COL1
cont>   EDIT STRING '+(9)';
SQL> SELECT COL1 FROM TEMP;
      COL1
    +53333
    -53333
</pre></td></tr>
</table>

(continued on next page)

Table 2–21 (Cont.) Numeric Floating Insertion Edit String Characters

Edit String Character	Description

–	If more than one minus sign (–) is specified to the left of the other edit string characters, suppresses any leading zeros in the same position as minus signs. If the value of the column is negative, displays a minus sign to the left of the leftmost digit.
	
	`SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:`
	`SQL> -- Compare this with single - edit string character:`
	`SQL> ALTER TABLE TEMP ALTER COL1`
	`cont>   EDIT STRING '-(9)';`
	`SQL> SELECT COL1 FROM TEMP;`
	`      COL1`
	`     53333`
	`    -53333`

Table 2–22 lists the floating-point, null value, and missing value edit string characters.

Table 2–22 Floating-Point, Null Value, and Missing Value Edit String Characters

Character Type	Edit String Character	Description

Floating-Point Edit String	E	The *E* divides the edit string into two parts for floating-point or scientific notation. The first part controls display of the mantissa, and the second part controls display of the exponent.
		
		`SQL> -- COL1 is INTEGER and contains the values 53333 and -53333:`
		`SQL> ALTER TABLE TEMP ALTER COL1`
		`cont>   EDIT STRING '+9.9(4)E+9';`
		`SQL> SELECT COL1 FROM TEMP;`
		`       COL1`
		` +5.3333E+4`
		` -5.3333E+4`

(continued on next page)

Table 2–22 (Cont.) Floating-Point, Null Value, and Missing Value Edit String Characters

Character Type	Edit String Character	Description

Null Value Missing Value	?	The question mark (?) denotes the beginning of a quoted string displayed when the column value is null (in SQL) or matches the value specified in a MISSING VALUE clause (in DATATRIEVE). If the column value is not null, the edit string preceding the question mark controls display of the value.
		
		`SQL> ALTER TABLE EMPLOYEES ALTER MIDDLE_INITIAL`
		`cont>   EDIT STRING 'X.?''No middle initial';`
		`SQL> SELECT MIDDLE_INITIAL FROM EMPLOYEES LIMIT TO 10 ROWS;`
		` MIDDLE_INITIAL`
		` D.`
		` G.`
		` P.`
		` O.`
		` M.`
		` No middle initial`
		` I.`
		` No middle initial`
		` A.`
		` E.`
		`10 rows selected`

Table 2–23 lists the date replacement edit string characters for the DATE VMS data type.

Table 2–23 Date Replacement Edit String Characters

Edit String Character	Description

D	Replaces each *D* with the corresponding digit of the day of the month. Put no more than two *D*s in a date edit string; the use of *DD* is recommended.
	
	`SQL> ALTER TABLE EMPLOYEES ALTER BIRTHDAY`
	`cont>   EDIT STRING 'DD-MMM-YYYY';`
	`SQL> SELECT BIRTHDAY FROM EMPLOYEES LIMIT TO 2 ROWS;`
	` BIRTHDAY`
	` 15-May-1954`
	` 12-Jan-1923`
	`2 rows selected`

(continued on next page)

Table 2–23 (Cont.) Date Replacement Edit String Characters

Edit String Character	Description
H	Replaces each *H* with the corresponding digit of the hour of the day in 12-hour notation.
R	Replaces each *R* with the corresponding digit of the hour of the day in 24-hour notation.
P	Replaces each *P* with the minute of the hour.
Q	Replaces each *Q* with the second of the minute.

M	Replaces each *M* with the corresponding letter of the name of the month. An edit string of *M*(9) prints the entire name of the month.
	`SQL> ALTER TABLE EMPLOYEES ALTER BIRTHDAY`
	`cont>   EDIT STRING 'M(9)BDD,BYYYY';`
	`SQL> SELECT BIRTHDAY FROM EMPLOYEES LIMIT TO 2 ROWS;`
	` BIRTHDAY`
	` March 28, 1947`
	` May 15, 1954`
	`2 rows selected`

N	Replaces each *N* with a digit of the number of the month. Put no more than two *N*s in a date edit string; the use of *NN* is recommended.
	`SQL> ALTER TABLE EMPLOYEES ALTER BIRTHDAY`
	`cont>   EDIT STRING 'NN/DD/YYYY';`
	`SQL> SELECT BIRTHDAY FROM EMPLOYEES LIMIT TO 2 ROWS;`
	` BIRTHDAY`
	`  5/15/1954`
	`  1/12/1923`
	`2 rows selected`

Y	YY represents the year of the century and YYYY represents the year and century. This release supports YYYYY for dates beyond the year 9999. For example, YYYYNNDD can represent 19990114.

J	Replaces each *J* with the corresponding digit of the Julian calendar date. Put no more than three *J*s in a date edit string; the use of *JJJ* is recommended.
	`SQL> ALTER TABLE EMPLOYEES ALTER BIRTHDAY`
	`cont>   EDIT STRING`
	`cont>          'M(9)BDD'' is the ''JJJ''th day of ''YYYY';`
	`SQL> SELECT BIRTHDAY FROM EMPLOYEES LIMIT TO 2 ROWS;`
	` BIRTHDAY`
	` March 28 is the 087th day of 1947`
	` May 15 is the 135th day of 1954`
	`2 rows selected`

(continued on next page)

Table 2–23 (Cont.) Date Replacement Edit String Characters

Edit String Character	Description

W	Replaces each *W* with the corresponding letter from the day of the week. An edit string of *W*(9) prints the entire day. Put no more than 9 *W*s in a date edit string.
	`SQL> ALTER TABLE EMPLOYEES ALTER BIRTHDAY`
	`cont>   EDIT STRING 'W(9),BM(9)BDD,BYYYY';`
	`SQL> SELECT BIRTHDAY FROM EMPLOYEES LIMIT TO 2 ROWS;`
	` BIRTHDAY`
	` Friday, March 28, 1947`
	` Saturday, May 15, 1954`
	`2 rows selected`

B	Replaces each *B* with a blank space in that character position.
/	Inserts a slash (/) in that character position.
-	Inserts a hyphen (-) in that character position.
:	Inserts a colon (:) in that character position.
.	Inserts a period (.) in that character position.
%	Inserts a percent Meridian indicator (%) in that character position. This string defaults to "AM" before noon and "PM" after noon.
* (asterisk)	Replaces each * with fractions of a second. For example, RR:PP:QQ.** represents 22:34:45.56.

If you specify an edit string incompatible with a column, SQL displays question marks when it retrieves the column values.

```
SQL> ALTER TABLE EMPLOYEES ALTER ADDRESS_DATA_1 EDIT STRING '99999';
SQL> SELECT ADDRESS_DATA_1 FROM EMPLOYEES LIMIT TO 2 ROWS;
 ADDRESS_DATA_1
 ??????????????
 ??????????????
2 rows selected
```

Usage Notes

- On OpenVMS systems the SET LANGUAGE statement can be used to select an alternate national language environment. Some languages define an alternate translation for the meridian indicators using the logical names LIBMI_U, LIBMI_L, and LIB$MI_C defined in the language specific logical name table. For instance, the Swedish translations are "FM" and

"EM" and are defined in the LNM$LANGUAGE_SWEDISH logical name table.

```
$ show logical/table=LNM$LANGUAGE_SWEDISH LIB$MI*

(LNM$LANGUAGE_SWEDISH)

  "LIB$MI_C" = "Fm"
        = "Em"
  "LIB$MI_L" = "fm"
        = "em"
  "LIB$MI_U" = "FM"
        = "EM"
```

SQL will attempt to use the translation of LIB$MI_U, then LIB$MI_L and finally LIB$MI_C before defaulting to the strings "AM" and "PM" if no translation is available for the current language.

- If the edit characters are narrower than required to display the field value then trailing digits or letters will be truncated. Excess editing characters are ignored. Therefore, 'RRRR:PP' is treated as though 'RR:PP' were specified.

Note

Future versions of Oracle Rdb may support more precision for the fractional seconds edit string. Therefore, do not use more than two asterisks because the wider field width may be interpreted differently in a future version.

- Oracle Rdb automatically trims leading zeros from the first numeric field in the output, and any numeric field following a space character. The space may appear in a quoted literal or the space formatting character (B). The year (Y) and fractional seconds (*) format fields are never trimmed of leading zeros.

Example

The following example shows these new edit characters.

```
SQL> set default date format 'SQL92';
SQL> set quoting rules 'SQL92';
SQL>
SQL> create table T (dt timestamp(2)
cont>                   edit string 'YYYY-NN-DDBRR:PP:QQ.**');
SQL> insert into T value (current_timestamp);
1 row inserted
SQL>
SQL> select dt from T;
 DT
 1999-01-16 10:17:12.63
1 row selected
SQL>
SQL> select dt as "Just Time" edit using 'HH:PPB%%' from T;
 Just Time
 10:17 AM
1 row selected
SQL>
SQL> set language SWEDISH;
SQL>
SQL> select dt as "Just Time" edit using 'HH:PPB%%' from T;
 Just Time
 10:17 FM
1 row selected
```

2.6 Value Expressions

A **value expression** is a symbol or string of symbols used to represent or calculate a single value. When you use a value expression in a statement, SQL retrieves or calculates the value associated with the expression and uses that value when executing the statement.

Value expressions are also called **scalar expressions** or expressions.

There are several different types of value expressions:

- A literal directly specifies a value. See Section 2.4 for more information.
- A parameter represents a value in a host language program or in an SQL module. See Section 2.2.13 for more information.
- A column name represents a value contained in table rows. See Section 2.2.4 for details on specifying value expressions with column names.
- A column select expression used as a value expression specifies a one-value result table. See Section 2.8.2 for more information.
- A built-in function calculates values based on input value expressions. See Section 2.6.2 for details.

SQL built-in functions include functions such as CAST, CURRENT_USER, and TRIM. For a complete list of built-in functions, see Section 2.6.2.

- An aggregate function calculates a single value for a collection of rows in a result table. See Section 2.6.3 for details.

  SQL aggregate functions are:

  – AVG
  – COUNT
  – MAX
  – MIN
  – STDDEV, STDDEV_SAMP, STDDEV_POP
  – SUM
  – VARIANCE, VAR_SAMP, VAR_POP

- SQL functions (CONCAT, CONVERT, DECODE, and SYSDATE) have been added to the Oracle Rdb SQL interface for convergence with Oracle SQL. See the *Oracle Server SQL Language Reference Manual* for more information.
- The DBKEY or ROWID keyword represents the value of an internal pointer called a database key to a table row. The ROWID keyword is a synonym to the DBKEY keyword. See Section 2.6.5 for more information.
- A character value expression represents a value that belongs to the CHAR, CHARACTER, VARCHAR, LONG VARCHAR, NCHAR, or NCHAR VARYING data type. You can link two character value expressions together using the concatenation operator (| |).
- You can also combine certain value expressions with arithmetic operators to form a value expression.
- A substring specifies a portion of a character value expression that you can manipulate using arithmetic operators.
- A conditional expression is a form of the value expression that allows applications to return alternative information within an expression. See Section 2.6.8 for details.

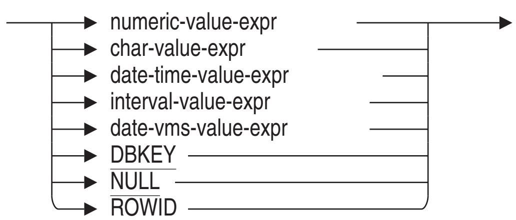
  Conditional expressions are:

  – CASE
  – COALESCE (or NVL)
  – DECODE

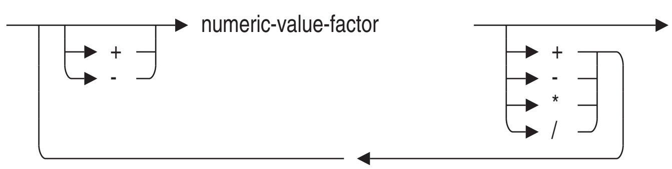
– GREATEST
– LEAST
– NULLIF
– NVL2

The following syntax diagrams show the format of an SQL value expression:

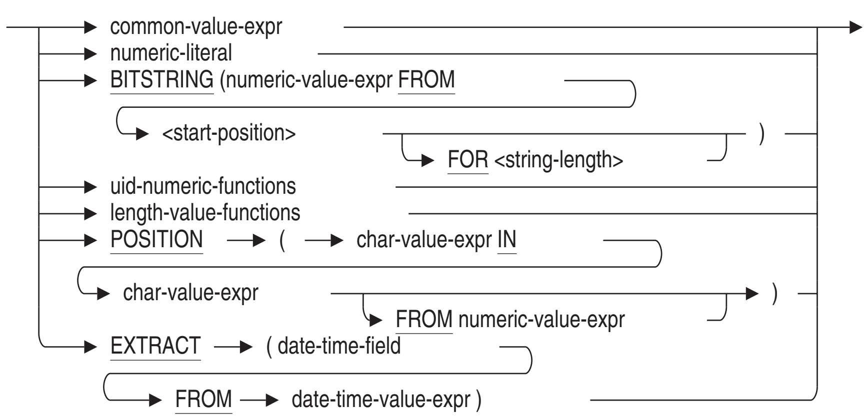
value-expr =



numeric-value-expr =



numeric-value-factor =



common-value-expr =

```
──┬─► <column-name> ─────────────────────────────┬─►
  ├─► <parameter> ───────────────────────────────┤
  ├─► <qualified-parameter> ─────────────────────┤
  ├─► <variable> ────────────────────────────────┤
  ├─► (col-select-expr) ─────────────────────────┤
  ├─► CAST ─► ( ─► value-expr ─► AS ┬► data-type ──────┬► ) ─┤
  │                                 └► <domain-name> ──┘     │
  ├─► VALUE ─────────────────────────────────────┤
  ├─► aggregate-function ────────────────────────┤
  ├─► conditional-expr ──────────────────────────┤
  ├─► function-invocation ───────────────────────┤
  └─► (value-expr) ──────────────────────────────┘
```

aggregate-function =

```
──┬─► COUNT (*) ────────────────────────────────────────────┬┬───────────────────┬─►
  ├─► COUNT ───────┬─► ( ┬────────────┬─► value-expr ─► ) ──┤└─► filter-clause ──┘
  ├─► AVG ─────────┤     ├─► ALL ─────┤                     │
  ├─► MIN ─────────┤     └─► DISTINCT ┘                     │
  ├─► MAX ─────────┤                                        │
  ├─► SUM ─────────┤                                        │
  ├─► STDDEV ──────┤                                        │
  ├─► VARIANCE ────┘                                        │
  ├─► STDDEV_POP ──┬─► ( ─► numeric-value-expr ─► ) ────────┘
  ├─► STDDEV_SAMP ─┤
  ├─► VAR_POP ─────┤
  └─► VAR_SAMP ────┘
```

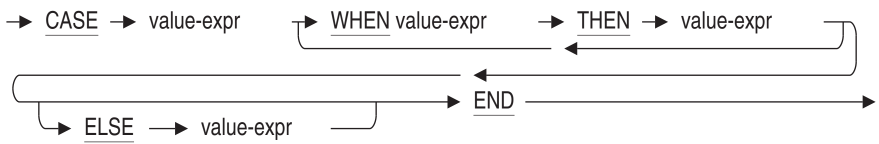
filter-clause =

```
──┬───────────────────────────────────────────┬─►
  └─► ( ─► WHERE ─► predicate ─► ) ───────────┘
```

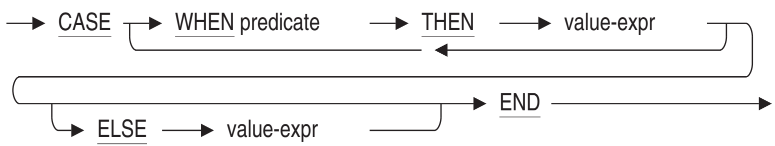
conditional-expr =

```
──┬─► NULLIF ─► ( ─► value-expr ─► , ─► value-expr ─► ) ──────┬─►
  ├─► COALESCE ─┬─► ( ─► value-expr ┬─► , ─► value-expr ┬─► ) ─┤
  ├─► NVL ──────┤                   └─────────◄─────────┘      │
  ├─► GREATEST ─┤                                              │
  ├─► LEAST ────┘                                              │
  ├─► NVL2 (value-expr, value-expr, value-expr) ───────────────┤
  ├─► ABS (value-expr) ────────────────────────────────────────┤
  ├─► simple-case-expr ────────────────────────────────────────┤
  └─► searched-case-expr ──────────────────────────────────────┘
```

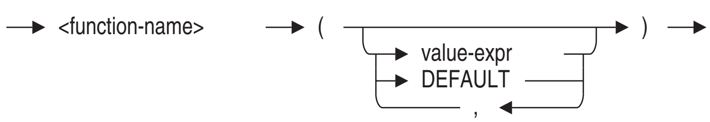
simple-case-expr =



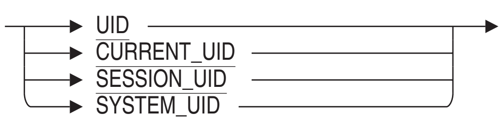
searched-case-expr =



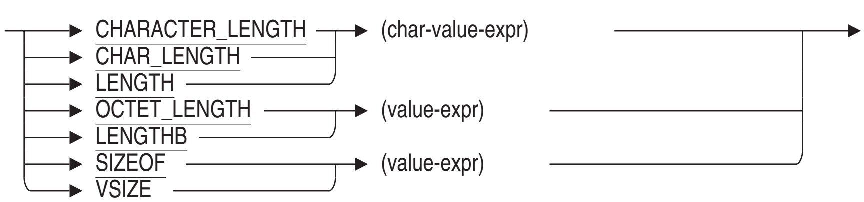
function-invocation =



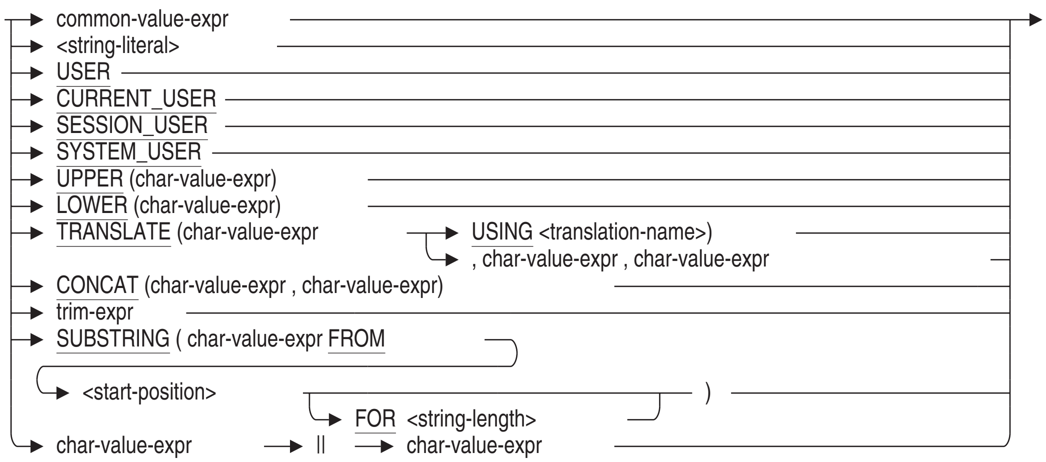
uid-numeric-functions =



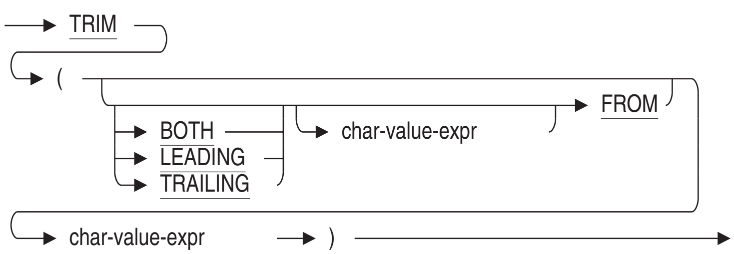
length-value-functions =



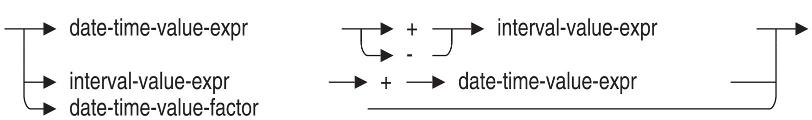
char-value-expr =



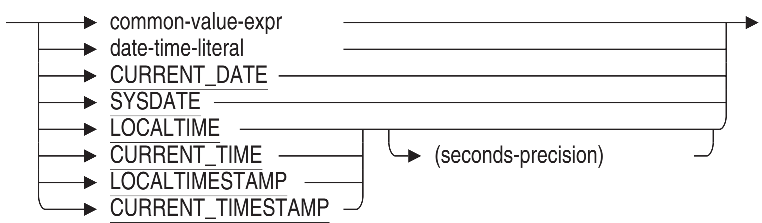
trim-expr =



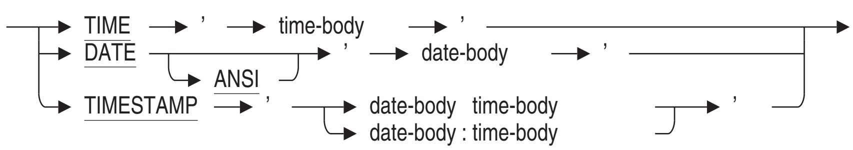
date-time-value-expr =



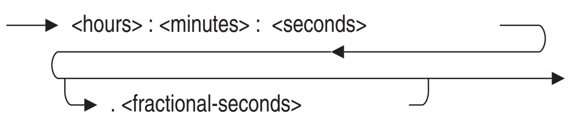
date-time-value-factor =



date-time-literal =

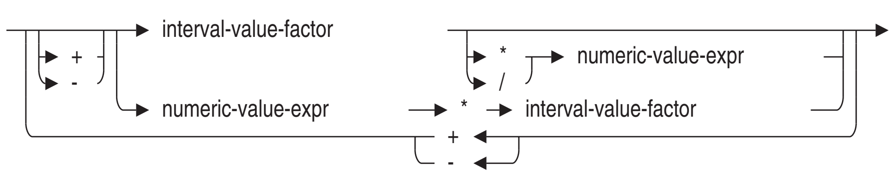


time-body =



date-body =

interval-value-expr =



interval-value-factor =

date-vms-value-expr =

For information regarding date-time data types, see Section 2.3.2.

The rest of this section describes functions, database keys, arithmetic expressions, and conditional expressions.

2.6.1 NULL Keyword Used as an Expression

The NULL keyword specifies the null value. When assigned to a column of any data type, the NULL keyword forces the column to be set to null because the NULL keyword has no data type. For example:

```
SQL> -- List all employees in the database and all potential employees
SQL> --
SQL> SELECT employee_id, last_name, first_name
cont> FROM employees
cont> UNION
cont> SELECT NULL, last_name, first_name
cont> FROM candidates;
   .
   .
   .
 00418        Blount          Peter
 00435        MacDonald       Johanna
 00471        Herbener        James
 NULL         Boswick         Fred
 NULL         Schwartz        Trixie
 NULL         Wilson          Oscar
103 rows selected
```

The NULL keyword is distinct from the IS NULL predicate which tests for null values of an expression. When testing an expression for null, using equality with NULL (for example, a = NULL) is not productive as it is never true. (See Table 2–29 and the note following the table.) Use the IS NULL predicate when testing an expression as shown in the following example:

```
SQL> SELECT e.last_name, j.job_end
cont> FROM employees e, job_history j
cont> WHERE e.employee_id = j.employee_id
cont> AND j.job_end IS NULL;
 E.LAST_NAME      J.JOB_END
 Smith            NULL
 O'Sullivan       NULL
 Hastings         NULL
   .
   .
   .
```

In some cases, the NULL keyword may have a data type of CHAR(31) when used in a query that requires a data type; such as in arithmetic and character expressions. In such queries, the assumption of the CHAR data type may cause an incompatibility error. If this occurs, use the CAST function (for example, CAST (NULL AS data-type)) to change NULL to a compatible data type for the query.

2.6.2 Built-In Functions

Built-in functions calculate values based on specified value expressions. Built-in functions are sometimes called **scalar functions**.

Table 2–24 describes these functions and the calculated value.

Table 2–24 Built-In Functions

Function Name	Calculated Value
CHARACTER_LENGTH CHAR_LENGTH LENGTH	Returns the length, in characters, of a value expression.
OCTET_LENGTH LENGTHB	Returns the length, in octets, of a value expression.
SIZEOF VSIZE	Returns the length or storage width of a value expression of any data type.

(continued on next page)

Table 2–24 (Cont.) Built-In Functions

Function Name	Calculated Value
BITSTRING	Extracts selected bits from a binary data value.
CAST	Converts a value expression to another data type.[1]
UPPER	Converts all lowercase characters in a value expression to uppercase characters.
LOWER	Converts all uppercase characters in a value expression to lowercase characters.
TRANSLATE	Translates a character value expression from one character set to another compatible character set.
SUBSTRING	Returns a portion of a character value expression.
EXTRACT	Extracts a single date-time field from a date-time value.
USER	Specifies the user name of the process that invokes interactive SQL or runs a program. USER is a synonym for CURRENT_USER.
CURRENT_USER	Returns the current active user name for a request.
SESSION_USER	Returns the current active session user name.
SYSTEM_USER	Returns the user name of the login process at the time of the database attach.
CURRENT_DATE	DATE data type value containing year, month, and day for date 'today'.
CURRENT_TIME	TIME data type value containing hour, minute, and second for time 'now'.
CURRENT_TIMESTAMP	TIMESTAMP data type value containing year, month, and day for date 'today' and hour, minute, and second for time 'now'.
LOCALTIME	Synonym for CURRENT_TIME
LOCALTIMESTAMP	Synonym for CURRENT_TIMESTAMP

[1]Applies to all data types except LIST OF BYTE VARYING.

(continued on next page)

Table 2–24 (Cont.) Built-In Functions

Function Name	Calculated Value
TRIM	Returns a character string minus a specified leading or trailing character (or both) of a value expression.
POSITION	Returns a numeric value that indicates the position of the search string in a source string.
Conditional expressions	See Section 2.6.8.
SYSDATE	Synonym for CURRENT_TIMESTAMP.
CONVERT	See Appendix G.
CONCAT	See Appendix G.
UID	A synonym for CURRENT_UID. UID is provided for Oracle and RDBMS compatibility.
CURRENT_UID	Returns a unique integer that represents the current user. This UID value is based on the AUTHORIZATION user or role for the currently executing stored routine, or the SESSION_USER if there is no current authorization.
SESSION_UID	Returns a unique integer that represents the session user.
SYSTEM_UID	Returns a unique integer that represents the system user.

The following sections describe these functions in more detail.

2.6.2.1 BITSTRING Function

The BITSTRING function extracts selected bits from a binary data value. This functionality is primarily intended to query the bit values stored in the RDB$FLAGS columns in the Rdb system table, but can also be used for user data.

BITSTRING accepts numeric and date/time values and processes them as bit arrays. The first (least significant) bit is numbered 1. The most significant bit depends on the data type.

- TINYINT has 8 bits
- SMALLINT has 16 bits
- INTEGER has 32 bits

- BIGINT, DATE, TIME, TIMESTAMP and INTERVAL types have 64 bits
- The numeric expression after the FOR and FROM keywords must be unscaled numeric values.

The following notes apply to usage of the BITSTRING function:

- If the numeric expression of the FOR clause is less than equal to zero then it will be assumed equal to 1.
- If the FOR clause is omitted it will default to a value that includes all remaining bits of the source value.
- If the FOR clause specifies a larger value than the number of bits remaining in the source then will only return the remaining bits.

Example: Using the BITSTRING function

Bit 1 in the RDB$FLAGS column of RDB$RELATIONS indicates that the table is a view. This example uses this query to fetch the names of all user defined views in the PERSONNEL database.

```
SQL> select rdb$relation_name
cont> from rdb$relations
cont> where rdb$system_flag = 0 and
cont>     bitstring (rdb$flags from 1 for 1) = 1;
 RDB$RELATION_NAME
 CURRENT_JOB
 CURRENT_SALARY
 CURRENT_INFO
3 rows selected
SQL>
```

2.6.2.2 CAST Function

The CAST function converts a value expression to another data type. The source and target columns can be of any data type except LIST OF BYTE VARYING.

If you convert to an INTERVAL data type, you must specify a single interval qualifier field, and the source must be a numeric value (fixed or floating) or a compatible INTERVAL data type. For information on interval qualifiers, see Section 2.3.2.

You can also convert from a single interval qualifier field to a numeric type (fixed or floating).

If you convert a TIMESTAMP literal using the CAST function, SQL puts a separating space character (SQL92) between the date-body and the time-body of the TIMESTAMP literal. For more information on TIMESTAMP literals, see Section 2.4.3.

The CAST function allows you to convert host language variables into date-time values. You can also use the CAST function to express dates in VMS format as ANSI format dates (using the syntax CAST(date-vms-value-expr AS DATE ANSI)) to do date arithmetic using DATE VMS data.

Examples: Using the CAST function

Example 1: Using the CAST function to list the number of months since the last salary raise in descending order for employees whose salary is above $50,000

```
SQL> CREATE VIEW MY_VIEW3
cont>   (TODAYS_DATE, SALARY_START, MONTHS_SINCE_RAISE)
cont> AS SELECT CURRENT_DATE, SALARY_START,
cont>   EXTRACT(MONTH FROM
cont>     (CURRENT_DATE - CAST(SH.SALARY_START AS DATE ANSI)) MONTH)
cont> FROM EMPLOYEES E, SALARY_HISTORY SH
cont> WHERE (E.EMPLOYEE_ID = SH.EMPLOYEE_ID) AND
cont>   (SALARY_AMOUNT >= 50000) AND
cont>   (SALARY_END IS NULL)
cont> ORDER BY 3 DESC;
SQL> SELECT * FROM MY_VIEW3;
 TODAYS_DATE   SALARY_START             MONTHS_SINCE_RAISE
 1993-12-02    12-MAR-1982 00:00:00.00                 141
 1993-12-02    10-MAR-1982 00:00:00.00                 141
 1993-12-02     6-APR-1982 00:00:00.00                 140
 1993-12-02    23-APR-1982 00:00:00.00                 140
 1993-12-02    18-MAY-1982 00:00:00.00                 139
   .
   .
   .
 1993-12-02     3-JAN-1983 00:00:00.00                 131
 1993-12-02    14-JAN-1983 00:00:00.00                 131
25 rows selected
```

Example 2: Using the CAST function to convert average salary information from scientific notation

```
SQL> --
SQL> -- First, without CAST, average is returned in floating-point
SQL> -- scientific notation.
SQL> --
SQL> CREATE VIEW MY_VIEW2 (DEPARTMENT_CODE, TOTAL_SALARY,
cont> AVERAGE_SALARY)
cont> AS SELECT DEPARTMENT_CODE, SUM(SALARY_AMOUNT),
cont> AVG(SALARY_AMOUNT)
cont>      FROM JOB_HISTORY JH,SALARY_HISTORY SH
cont>    WHERE (JH.EMPLOYEE_ID = SH.EMPLOYEE_ID)
cont>      AND (JH.JOB_END IS NULL)
cont>      AND (SH.SALARY_END IS NULL)
cont>    GROUP BY DEPARTMENT_CODE
cont>      HAVING SUM (SALARY_AMOUNT) > 100000
cont> ORDER BY 2 DESC, DEPARTMENT_CODE;
SQL> SELECT * FROM MY_VIEW2;
 DEPARTMENT_CODE             TOTAL_SALARY            AVERAGE_SALARY
 ADMN                           525403.00    7.505757142857143E+004
 ELEL                           208299.00    2.603737500000000E+004
 PHRN                           192393.00    3.847860000000000E+004
 PERL                           158752.00    3.175040000000000E+004
 SUWE                           157429.00    3.935725000000000E+004
SQL> --
SQL> -- Using CAST, the AVERAGE_SALARY output is converted from scientific
SQL> -- notation to a more readable format.
SQL> --
SQL> CREATE VIEW MY_VIEW2 (DEPARTMENT_CODE, TOTAL_SALARY,
cont> AVERAGE_SALARY)
cont>  AS SELECT DEPARTMENT_CODE, SUM(SALARY_AMOUNT),
cont>     CAST(AVG(SALARY_AMOUNT) AS BIGINT(2))
cont>  FROM JOB_HISTORY JH,SALARY_HISTORY SH
cont>     WHERE (JH.EMPLOYEE_ID = SH.EMPLOYEE_ID)
cont>      AND (JH.JOB_END IS NULL)
cont>      AND (SH.SALARY_END IS NULL)
cont> GROUP BY DEPARTMENT_CODE
cont> HAVING SUM (SALARY_AMOUNT) > 100000
cont> ORDER BY 2 DESC, DEPARTMENT_CODE;
SQL> --
SQL> SELECT * FROM MY_VIEW2;
 DEPARTMENT_CODE             TOTAL_SALARY            AVERAGE_SALARY
 ADMN                           525403.00                  75057.57
 ELEL                           208299.00                  26037.38
 PHRN                           192393.00                  38478.60
 PERL                           158752.00                  31750.40
 SUWE                           157429.00                  39357.25
```

Example 3: Using the CAST function to convert employee identification numbers to integers

```
SQL> SELECT CAST(EMPLOYEE_ID AS INTEGER) FROM EMPLOYEES LIMIT TO 1 ROW;

         164
1 row selected
```

2.6.2.3 CHARACTER_LENGTH Function

The CHARACTER_LENGTH (CHAR_LENGTH or LENGTH) function calculates the length of a value expression of any data type.

If the result of the value expression is a character data type, the CHARACTER_LENGTH function returns the length, in characters, of the character string. (Remember that the length of a character can be one or more octets.) If the result of the value expression is NULL, the function returns a null value. You can use CHAR_LENGTH or LENGTH as an alternative for CHARACTER_LENGTH.

Examples: Using the CHARACTER_LENGTH function

Example 1: Using the CHARACTER_LENGTH function to calculate the number of characters in the values in CHAR and VARCHAR columns

```
SQL> -- Because the column LAST_NAME is defined as CHAR(14), a fixed-length
SQL> -- data type, SQL pads the values in the column with blanks.  The
SQL> -- following statement returns the same value for all the rows.
SQL> --
SQL> SELECT CHARACTER_LENGTH(LAST_NAME), LAST_NAME
cont>       FROM EMPLOYEES LIMIT TO 3 ROWS;
             LAST_NAME
          14   Ames
          14   Andriola
          14   Babbin
3 rows selected
SQL> --
SQL> -- Because the column CANDIDATES_STATUS is defined as VARCHAR, a
SQL> -- varying-length data type, SQL does not pad the column with blanks.
SQL> --
SQL> SELECT CHARACTER_LENGTH(CANDIDATE_STATUS) FROM CANDIDATES;

          63
          69
          46
3 rows selected
```

Example 2: Using the CHARACTER_LENGTH function with multi-octet character sets

```
SQL> ! Using the COLOURS table in the MIA_CHAR_SET sample program
SQL> ! included with this release, select data specifying the
SQL> ! CHARACTER_LENGTH function
SQL> !
SQL> SELECT CHARACTER_LENGTH(ENGLISH), ENGLISH,
cont>       CHARACTER_LENGTH(FRENCH), FRENCH,
cont>       CHARACTER_LENGTH(JAPANESE), JAPANESE
cont> FROM COLOURS
cont> LIMIT TO 3 ROWS;
             ENGLISH                 FRENCH                  JAPANESE
          8   Yellow           8   Jaune              4   黄
          8   Black            8   Noir               4   黒
          8   Blue             8   Bleu               4   青
3 rows selected
```

2.6.2.4 CURRENT_DATE Function

The CURRENT_DATE function returns a DATE data type value (ANSI format) containing year, month, and day for date 'today'. You can specify an optional fractional-seconds precision for CURRENT_DATE.

Example: Using the CURRENT_DATE function

The following example shows how a site with an Oracle Rdb database might use the CURRENT_DATE function to determine employee ages. You must use the CAST function to convert the DATE column BIRTHDAY from VMS to ANSI format to use it with the ANSI format CURRENT_DATE function.

```
SQL> ATTACH FILENAME 'corporate_data';
SQL> SET SCHEMA 'ADMINISTRATION.PERSONNEL';
SQL> CREATE VIEW AGE (LAST_NAME, FIRST_NAME, BIRTHDAY, AGE)
cont>       AS SELECT LAST_NAME, FIRST_NAME, BIRTHDAY,
cont>        (CURRENT_DATE - CAST(BIRTHDAY AS DATE ANSI)) YEAR TO MONTH
cont>       FROM EMPLOYEES ORDER BY BIRTHDAY ASC LIMIT TO 10 ROWS;
SQL> --
SQL> -- A SELECT statement displays the ten oldest employees.
SQL> SELECT * FROM AGE;
 LAST NAME       FIRST_NAME       BIRTHDAY         AGE
 O'Sullivan      Rick             12-Jan-1923       68-06
 Clairmont       Rick             23-Dec-1924       66-07
 Nash            Walter           19-Jan-1925       66-06
 Kinmonth        Louis             7-Apr-1926       65-03
 Bartlett        Dean              5-Mar-1927       64-06
 Johnson         Bill             13-Apr-1927       64-03
 Herbener        James            28-Oct-1927       63-09
 Babbin          Joseph           12-Dec-1927       63-07
 Ziemke          Al               27-Oct-1928       62-09
 Reitchel        Charles          13-Dec-1928       62-07
10 rows selected
SQL>
```

2.6.2.5 CURRENT_TIME and LOCALTIME Functions

The CURRENT_TIME function returns a TIME data type value containing hours, minutes, and seconds for time `'now'`.

You can specify a fractional precision between 0 and 2 for the seconds returned by CURRENT_TIME. The fractional-seconds precision is a number that designates the number of digits returned in the field. For example, a fractional precision of 2 means that seconds are returned as hundredths of a second (2 digits beyond the decimal point), while a fractional precision of 1 means that only tenths of a second are returned (1 digit beyond the decimal point).

Example 1: The following example shows how to create a domain of data type TIME and insert the CURRENT_TIME into the column:

```
SQL> CREATE DOMAIN END_TIME_DOM IS TIME;
SQL> CREATE TABLE HOURS_WORKED (END_TIME END_TIME_DOM);
SQL> INSERT INTO HOURS_WORKED (END_TIME) VALUES (CURRENT_TIME);
1 row inserted
SQL> SELECT * FROM HOURS_WORKED;
 END_TIME
 15:03:07
1 row selected
```

You can specify a current default for a time or timestamp field with nondefault fractional-seconds precision, as shown in the following example:

Example 2: In this example, an error results when the user specifies a fractional-seconds precision different from the current default:

```
SQL> CREATE DOMAIN Y TIME(2) DEFAULT CURRENT_TIME(1);
%SQL-F-DEFVALINC, You specified a default value for Y which is
inconsistent with its data type
SQL> CREATE DOMAIN Y TIME(1) DEFAULT CURRENT_TIME(1);
```

The LOCALTIME built-in function is a synonym for CURRENT_TIME, and is defined by the SQL:1999 database language standard.

2.6.2.6 CURRENT_TIMESTAMP and LOCALTIMESTAMP Functions

The CURRENT_TIMESTAMP function returns a TIMESTAMP data type value containing year, month, and, day for date 'today' and hours, minutes, and seconds for time 'now'.

As in CURRENT_TIME, you can specify a fractional precision between 0 and 2 for the seconds returned by CURRENT_TIMESTAMP. The fractional-seconds precision is a number that designates the number of digits returned in the field.

The CURRENT_TIMESTAMP data type can be either DATE VMS or DATE ANSI format. Date-time arithmetic is not allowed with DATE VMS columns. A DATE VMS format CURRENT_TIMESTAMP specifies the day, month, and year of the current date and the hours, minutes, and seconds of the current time. A DATE ANSI format CURRENT_TIMESTAMP specifies the year, month and day of the current date, followed by the hours, minutes, and seconds of the current time.

The LOCALTIMESTAMP built-in function is a synonym for CURRENT_TIMESTAMP and is defined by the SQL:1999 database language standard.

Example: Using the CURRENT_TIMESTAMP function

Example 1: In the following example, SQL fills in a value for CURRENT_TIMESTAMP every time an INSERT statement is executed on ORDER_TABLE2:

```
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP(1) DEFAULT CURRENT_TIMESTAMP(1);
SQL> CREATE TABLE ORDER_TABLE2
cont> (PART_NUM       INT,
cont>  ORDER_LOGGED   LOGGING_DATE,
cont>  DELIVERY_DATE  TIMESTAMP(1),
cont>  TIME_TO_DELIVER
cont>   COMPUTED BY (DELIVERY_DATE - ORDER_LOGGED) DAY(6) TO MINUTE,
cont>  SLOW_DELIVERY
cont>   COMPUTED BY (EXTRACT(DAY FROM
cont>     (DELIVERY_DATE - ORDER_LOGGED) DAY(6)) - 30)
cont> );
```

Example 2: In the following example, SQL issues an error message because the CURRENT_TIMESTAMP data type uses the VMS format by default, and the TIMESTAMP data type uses the ANSI format:

```
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
%SQL-F-DEFVALINC, You specified a default value for LOGGING_DATE which is
inconsistent with its data type
```

SQL provides several ways to change DATE and CURRENT_TIMESTAMP data to ANSI format:

- The statement SET DIALECT 'SQL99'
- The statement SET DEFAULT DATE FORMAT
- The precompiler DEFAULT DATE FORMAT clause in a DECLARE MODULE statement embedded in a program
- The module language DEFAULT DATE FORMAT clause in a module file

Example 3: The following example shows the DATE VMS and DATE ANSI output formats for CURRENT_TIMESTAMP:

```
SQL> ATTACH 'FILENAME corporate_data';
SQL> SHOW ANSI DATE
DATE data type equates to DATE VMS
SQL> SELECT CURRENT_TIMESTAMP FROM DAILY_HOURS LIMIT TO 1 ROW;

 15-AUG-1991 10:40:52.83
1 row selected
SQL> SET DEFAULT DATE FORMAT 'SQL99';
SQL> SHOW ANSI DATE
DATE data type equates to DATE ANSI
SQL> SELECT CURRENT_TIMESTAMP FROM DAILY_HOURS LIMIT TO 1 ROW;
```

```
 1991-08-15 10:41:02.52
1 row selected
```

You must use the SET DEFAULT DATE FORMAT statement before creating domains or tables. You cannot use this statement to modify the data type after you created a database definition.

The CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP keywords are accessible from anywhere that an expression is allowed in Oracle Rdb.

Example 4: The following is an example of an SQL statement that inserts the date into the JOB_START column of a JOB_HISTORY table:

```
SQL> INSERT INTO JOB_HISTORY (JOB_START  . . .  )
cont> VALUES (CURRENT_TIMESTAMP,  . . .  );
```

If you use the CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP keyword more than once within a statement, it retains the same value for the date and time.

Example 5: This query requires that the difference of absolute dates be calculated and the year component is then selected (and printed) from the calculated interval:

```
SQL> SELECT  FIRST_NAME, LAST_NAME, ' is ',
cont> EXTRACT(YEAR FROM (CURRENT_TIMESTAMP - BIRTHDAY) YEAR),
cont> ' years old'
cont> FROM    EMPLOYEES;
```

Example 6: The trigger in the following example records a history of updates to the EMPLOYEES table. The HISTORY table in the example contains the date and time of any updates to table rows containing employee birthdays, the name of the user making the updates, and the employee ID number of the updated rows:

```
SQL> -- Create a new table for the trigger.
SQL> CREATE TABLE HISTORY
cont> ("DATE" DATE,
cont>  USER_NAME CHAR(14),
cont>  UPDATED_ID CHAR(5));
SQL> --
SQL> CREATE TRIGGER EMP_UPD_TRIG AFTER UPDATE ON EMPLOYEES
cont> (INSERT INTO HISTORY ("DATE", USER_NAME, UPDATED_ID)
cont> VALUES (CURRENT_DATE, USER, EMPLOYEE_ID))
cont> FOR EACH ROW;
```

In general, all triggers executed as part of a statement receive the same timestamp. The timestamp is the time that the statement is executed.

Example 7: You can also set the date to correspond to the DEFAULT clause of the CURRENT_TIMESTAMP keyword. In that case, SQL fills in a value for CURRENT_TIMESTAMP every time an INSERT statement is executed:

```
SQL> CREATE TABLE TIMESTAMP_TABLE
cont>   (LOG_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
cont>    USER_NAME CHAR(14) DEFAULT USER,
cont>    UPDATED_ID CHAR(5));
SQL> --
SQL> CREATE TRIGGER EMP_UPD_TRIG AFTER UPDATE ON EMPLOYEES
cont> (INSERT INTO HISTORY (UPDATED_ID)
cont> VALUES (EMPLOYEE_ID))
cont> FOR EACH ROW;
```

2.6.2.7 CURRENT_UID Function

The CURRENT_UID function returns a unique integer that represents the current user. This UID value is based on the AUTHORIZATION user or role for the currently executing stored routine, or the SESSION_USER if there is no current authorization.

2.6.2.8 CURRENT_USER Function

The CURRENT_USER function returns the current active user name for a request.

If a definer's rights request is executing, the CURRENT_USER function returns the rights identifier of the module definer. If a definer's rights request is not executing, CURRENT_USER returns the session user name, if it exists. Otherwise, CURRENT_USER returns the system user name. See Section 2.2.2 for more information.

The resulting data type is CHAR(31).

The CURRENT_USER function does not return the definer's user name of a trigger.

Example: Using the CURRENT_USER function

Example 1: To allow users access only to the rows they inserted, create a view

```
SQL> CREATE VIEW SELECTIVE_EMPLOYEES_UPDATE AS
cont>   SELECT * FROM EMPLOYEES
cont>   WHERE USER_ID = CURRENT_USER
cont>   WITH CHECK OPTION CONSTRAINT MUST_HAVE_USER;
```

2.6.2.9 EXTRACT Function

The EXTRACT function returns a single date-time field expressed as an integer from a column of data type DATE, TIME, TIMESTAMP, or INTERVAL.

The date-time fields that EXTRACT can return are:

- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- WEEKDAY
- JULIAN
- WEEK_NUMBER
- YEAR_WEEK

The data type returned is a signed longword of scale 0, unless the date-time field is SECOND. If the SECOND field is selected, then the scale is set to 2.

If you specify WEEKDAY, you can only use the data types TIMESTAMP and DATE as the extract source. In all other cases, the extract source can be data type DATE, TIME, TIMESTAMP, or INTERVAL. If you specify WEEKDAY, then the EXTRACT function returns an integer representing the day of the week. (Monday is represented as day 1, Sunday as day 7.)

If the EXTRACT function is applied to a null value, it returns a null value.

The number of days since the first day of a year, called the Julian date, can be an important integer value to which programmers need direct access. The SQL EXTRACT function lets you determine the Julian date from column data defined with date-time data types.

The JULIAN keyword requires that the extract expression resolve to either the DATE ANSI or TIMESTAMP date-time data type. Value expressions that do not resolve to one of these particular data types will fail. For example, trying to extract the Julian date from an expression defined by the CURRENT_TIME data type results in the following SQL error message:

```
SQL> SELECT EXTRACT(JULIAN FROM CURRENT_TIME) FROM ACCOUNTING.DAILY_HOURS;
%RDB-F-CONVERT_ERROR, invalid or unsupported data conversion
-RDMS-E-EXT_JULIAN_TS, invalid type for EXTRACT JULIAN, must be DATE or
TIMESTAMP
SQL>
```

You cannot represent dates from the year 1858 using the JULIAN keyword in the EXTRACT function because JULIAN calculates from 1-January and the first date in 1858 is 18-November.

The options WEEK_NUMBER and YEAR_WEEK return the week number as defined by the International Standard ISO 8601:1988 "Data elements and interchange formats - Information interchange - Representation of dates and times".

WEEK_NUMBER is a number between 1 and 53 representing the week of the year (most years only have 52 weeks). A week starts on Monday and has most of its days falling in a specific year.

YEAR_WEEK is a variation of the WEEK_NUMBER that includes the year (including the century) in which the week logically falls. The values range from 185901 through 999952 (higher values are possible if dates are constructed with a year beyond 9999). The last two digits of the value are identical to the value returned by the WEEK_NUMBER option.

Examples: Using the EXTRACT function

Example 1: Using the EXTRACT function to find the employee with the longest record of service who is still employed by the company

```
SQL> CREATE VIEW MY_VIEW2
cont>       (LAST_NAME, TODAYS_DATE, JOB_START, MONTHS_EMPLOYED)
cont>    AS SELECT E.LAST_NAME, CURRENT_DATE, JH.JOB_START,
cont>       EXTRACT (MONTH FROM
cont>           (CURRENT_DATE - CAST(JH.JOB_START AS DATE ANSI)) MONTH)
cont>    FROM EMPLOYEES E, JOB_HISTORY JH
cont>    WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
cont>       AND
cont>       (CURRENT_DATE - CAST(JH.JOB_START AS DATE ANSI)) MONTH =
cont>       (SELECT
cont>           MAX ((CURRENT_DATE - CAST (JH.JOB_START AS DATE ANSI)) MONTH)
cont>        FROM JOB_HISTORY JH);
```

```
SQL> SELECT * FROM MY_VIEW2;
 LAST_NAME        TODAYS_DATE   JOB_START                 MONTHS_EMPLOYED
 Smith            1993-12-02     1-JUL-1975 00:00:00.00               221
 Nash             1993-12-02     1-JUL-1975 00:00:00.00               221
 Gray             1993-12-02     1-JUL-1975 00:00:00.00               221
 Peters           1993-12-02     1-JUL-1975 00:00:00.00               221
   .
   .
   .
 Ames             1993-12-02     1-JUL-1975 00:00:00.00               221
 Blount           1993-12-02     1-JUL-1975 00:00:00.00               221
43 rows selected
```

Example 2: Using the EXTRACT function to compute when ordered items are overdue

```
SQL> SET DEFAULT DATE FORMAT 'SQL92';
SQL> CREATE DOMAIN LOGGING_DATE TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
SQL> CREATE TABLE ORDER_TABLE
cont>  (ORDER_NUMBER     INT,
cont>   COMPANY_NAME     VARCHAR(40),
cont>   ORDER_LOGGED     LOGGING_DATE,
cont>   DELIVERY_DATE    DATE ANSI,
cont>   TIME_TO_DELIVER
cont>       COMPUTED BY (DELIVERY_DATE - ORDER_LOGGED) DAY(3) TO MINUTE.
cont>   SLOW_DELIVERY
cont>     COMPUTED BY (EXTRACT(DAY FROM (DELIVERY_DATE - ORDER_LOGGED)
cont>     DAY) - 30));
SQL> INSERT INTO ORDER_TABLE
cont>     (ORDER_NUMBER,
cont>      COMPANY_NAME,
cont>      ORDER_LOGGED,
cont>      DELIVERY_DATE)
cont>VALUES
cont>    (1,
cont>     'ABC INC.',
cont>     TIMESTAMP '1991-2-4 10:30:00.00',
cont>     DATE '1991-6-1'
cont>    );
1 row inserted
SQL> --
SQL> INSERT INTO ORDER_TABLE
cont>    (ORDER_NUMBER,
cont>     COMPANY_NAME,
cont>     DELIVERY_DATE)
cont> VALUES
cont>    (2,
cont>     'JJ ROOFING',
cont>     DATE '1991-5-1'
cont>    );
1 row inserted
SQL> --
```

```
SQL> SELECT ORDER_NUMBER, ORDER_LOGGED, DELIVERY_DATE FROM ORDER_TABLE;
 ORDER_NUMBER   ORDER_LOGGED                DELIVERY_DATE
            1   1991-02-04 10:30:00.000000   1991-06-01
            2   1991-04-18 09:06:05.630000   1991-05-01
2 rows selected
SQL> --
SQL> SELECT TIME_TO_DELIVER, SLOW_DELIVERY FROM ORDER TABLE
cont> WHERE SLOW_DELIVERY >= 0;
 TIME_TO_DELIVER          SLOW_DELIVERY
  116:13:30                          86
1 row selected
SQL> --
SQL> SELECT COMPANY_NAME,EXTRACT(WEEKDAY FROM ORDER_LOGGED) FROM ORDER_TABLE;
 COMPANY_NAME
 ABC INC.                                          1
 JJ ROOFING                                        4
2 rows selected
```

Example 3: Calculating the Julian date with the EXTRACT function

```
SQL> -- Attach to the multischema database corporate_data and define
SQL> -- a default catalog and schema setting.
SQL> --
SQL> ATTACH 'FILENAME corporate_data';
SQL> SET CATALOG 'ADMINISTRATION';
SQL> SET SCHEMA 'PERSONNEL';
SQL> --
SQL> -- Create view to show column heads for SELECT statement. The EXTRACT
SQL> -- function using the new JULIAN keyword calculates the Julian date
SQL> -- of an employee's birthday.
SQL> --
SQL> CREATE VIEW JULIAN_YEAR
cont>      (LAST_NAME, EMPLOYEE_ID, BIRTHDAY, JULIAN_DATE)
cont>      AS SELECT LAST_NAME, EMPLOYEE_ID, BIRTHDAY,
cont>      EXTRACT(JULIAN FROM BIRTHDAY)
cont>      FROM EMPLOYEES WHERE EMPLOYEE_ID = '00415';
SQL> SELECT * FROM JULIAN_YEAR;
LAST_NAME              EMPLOYEE_ID   BIRTHDAY     JULIAN_DATE
Mistretta              00415         1947-05-23           143
1 row selected
SQL> ROLLBACK;
```

The Julian date 143 represents the number of days from January 1, 1947 to May 23, 1947. (The EXTRACT function would have returned the Julian date 144 if the employee was born on the same day in the leap year of 1948.) You can try this example using the corporate_data multischema database from the Samples directory.

Example 4: Using the EXTRACT function with YEAR_NUMBER and YEAR_WEEK.

```
SQL> select dt,
cont>       extract (week_number from dt),
cont>       extract (year_week from dt)
cont> from week_sample
cont> order by dt;
 DT
 1859-01-07             1        185901
 1999-01-01            53        199853
 1999-01-04             1        199901
 1999-01-10             1        199901
 1999-12-31            52        199952
 2000-01-01            52        199952
 2000-01-03             1        200001
 2000-02-28             9        200009
 2000-02-29             9        200009
 2000-03-01             9        200009
 9999-12-31            52        999952
11 rows selected
```

2.6.2.10 LOWER Function

The LOWER function converts all uppercase characters in a value expression to lowercase characters. This function is useful to maintain consistency in value expressions in the database.

If the result of the value expression is NULL, the function returns a null value.

Example: Using the LOWER function

Use the LOWER function to convert the uppercase characters in DEPARTMENT_NAME to lowercase characters:

```
SQL> SELECT DEPARTMENT_NAME, LOWER(DEPARTMENT_NAME)
cont> FROM DEPARTMENTS
cont> LIMIT TO 3 ROWS;
 DEPARTMENT_NAME
 Corporate Administration         corporate administration
 Electronics Engineering          electronics engineering
 Large Systems Engineering        large systems engineering
3 rows selected
SQL>
```

When you use the LOWER function, SQL follows the rules of the character set for the value expression when converting characters to lowercase. For example, if the character set of the value expression is Hanzi and ASCII, SQL converts only the ASCII characters to lowercase. It does not convert the Hanzi characters.

2.6.2.11 OCTET_LENGTH Function

The OCTET_LENGTH (or LENGTHB) function calculates the length, in octets, of a value expression of any data type.

If the result of the value expression is NULL, the function returns a null value. Otherwise, the function returns the length, in octets, of the value expression. You can use LENGTHB as an alternative for OCTET_LENGTH.

Examples: Using the OCTET_LENGTH function

Example 1: Using the OCTET_LENGTH function to calculate the number of characters in the values in CHAR and VARCHAR columns

```
SQL> -- This example uses the personnel sample database.
SQL> -- Because the column LAST_NAME is defined as CHAR(14), a fixed-length
SQL> -- data type, SQL pads the values in the column with blanks.  The
SQL> -- following statement returns the same value for all the rows.
SQL> --
SQL> SELECT OCTET_LENGTH (LAST_NAME), LAST_NAME
cont>       FROM EMPLOYEES
cont>       LIMIT TO 3 ROWS;
               LAST_NAME
          14   Ames
          14   Andriola
          14   Babbin
3 rows selected
SQL> --
SQL> -- Because the column CANDIDATE_STATUS is defined as VARCHAR(255), a
SQL> -- varying-length data type, SQL does not pad the column with blanks.
SQL> --
SQL> SELECT OCTET_LENGTH (CANDIDATE_STATUS) FROM CANDIDATES;

          63
          69
          46
3 rows selected
SQL>
```

Example 2: Using the OCTET_LENGTH function with multi-octet character sets

```
SQL> ! Using the COLOURS table in the MIA_CHAR_SET sample program
SQL> ! included with this release, select data specifying the
SQL> ! OCTET_LENGTH function
SQL> !
SQL> SELECT OCTET_LENGTH(ENGLISH), ENGLISH,
cont>        OCTET_LENGTH(FRENCH), FRENCH,
cont>        OCTET_LENGTH(JAPANESE), JAPANESE
cont> FROM COLOURS
cont> LIMIT TO 3 ROWS;
               ENGLISH                  FRENCH                  JAPANESE
           8   Yellow               8   Jaune               8   黄
           8   Black                8   Noir                8   黒
           8   Blue                 8   Bleu                8   青
3 rows selected
```

2.6.2.12 POSITION Function

The POSITION function searches for a string in a character value expression. The first character value expression is also called a search string. The second character value expression is also called a source string. If the search string is located, the POSITION function returns a numeric value that indicates the position of the search string in the source string. The returned numeric value is the absolute position of the search string in the source string starting with 1. The match between the search string and the source string is case sensitive.

If the search string is not found in the source string, the POSITION function returns a zero (0) value. If any of the strings is NULL, the result is NULL.

The FROM clause of the POSITION function is an extension to the ANSI/ISO SQL standard and allows searching to begin from any location.

Examples: Using the POSITION function

Example 1: Using the POSITION function in a SELECT statement

```
SQL> SELECT COLLEGE_NAME,
cont> POSITION ('University' IN COLLEGE_NAME)
cont> FROM COLLEGES
cont> WHERE COLLEGE_NAME LIKE '_%University%';
 COLLEGE_NAME
 American University                10
 Drew University                     6
 Harvard University                  9
 Purdue University                   8
 Stanford University                10
 Yale University                     6
6 rows selected
```

Example 2: Using the POSITION function with the SUBSTRING clause

```
SQL> SELECT SUBSTRING (COLLEGE_NAME FROM 1 FOR
cont>                  POSITION ('University' IN COLLEGE_NAME) -1)
cont> FROM COLLEGES
cont> WHERE COLLEGE_NAME LIKE '_%University%';

 American
 Drew
 Harvard
 Purdue
 Stanford
 Yale
6 rows selected
```

Example 3: Using the POSITION function to find individual words. Because this example uses the TRACE statement, you must define the RDMS$DEBUG_FLAGS logical name to "Xt".

```
SQL> BEGIN
cont>   DECLARE :TXT VARCHAR(100);
cont>   DECLARE :RES VARCHAR(20);
cont>   DECLARE :ST, :EN INTEGER;
cont> --
cont>   SET :TXT = 'Some words and phrases';
cont> --
cont> --     Start at the beginning
cont> --
cont>   SET :ST = 1;
cont> --
cont> --     Loop over all the text looking for space delimiters
cont> --
cont>   WHILE :ST <= CHAR_LENGTH(:TXT)
cont>   LOOP
cont>     SET :EN = POSITION (' ' IN :TXT FROM :ST);
cont>     IF :EN = 0 THEN
cont> --
cont> --     No trailing spaces, so assume space after last character
cont> --
cont>       SET :EN = CHAR_LENGTH(:TXT) + 1;
cont>     END IF;
cont>     SET :RES = SUBSTRING(:TXT FROM :ST FOR :EN - :ST);
cont>     IF CHAR_LENGTH (TRIM (:RES)) > 0 THEN
cont> --
cont> --     Have a word to display
cont> --
cont>       TRACE 'Word: "', :RES, '"';
cont>     END IF;
cont> --
cont> --     Advance the start position
cont> --
cont>     SET :ST = :EN + 1;
```

```
cont>   END LOOP;
cont> END;
~Xt: Word: "Some                "
~Xt: Word: "words               "
~Xt: Word: "and                 "
~Xt: Word: "phrases             "
```

2.6.2.13 SESSION_UID Function

Returns a unique integer that represents the session user.

2.6.2.14 SESSION_USER Function

The SESSION_USER function returns the current active session user name.

If the session user name is not returned, the system user name is returned. The resulting data type is CHAR(31).

2.6.2.15 SIZEOF Function

The SIZEOF (or VSIZE) function calculates the maximum length, in octets, of a value expression of any data type. It returns the length or storage width of the value expression. Like OCTET_LENGTH, SIZEOF returns the number of eight-bit units (octets) rather than the number of characters if the expression yields a text string in a multi-byte character set. You can use VSIZE as an alternative for SIZEOF.

Examples: Using the SIZEOF function

Example 1: Using the SIZEOF function to calculate the number of maximum length of CHAR columns.

This example uses the personnel sample database. The column LAST_NAME is defined as CHAR(14), a fixed-length data type for which SQL pads the values in the column with blanks. The following statement returns the same value as if OCTET_LENGTH had been used in lieu of SIZEOF.

```
SQL> SELECT SIZEOF (LAST_NAME), LAST_NAME
cont>       FROM EMPLOYEES
cont>       LIMIT TO 3 ROWS;
               LAST_NAME
          14   Ames
          14   Andriola
          14   Babbin
3 rows selected
```

Example 2: Using the SIZEOF function to calculate the number of maximum length of VARCHAR columns.

The column CANDIDATE_STATUS is defined as VARCHAR(255), a varying-length data type for which SQL does not pad the column with blanks. Because SIZEOF returns the maximum size (storage width) of the expression, it returns the same value (255) for all rows. This is in contrast with OCTET_LENGTH, which returns the actual size of the data in each row.

```
SQL> SELECT SIZEOF (CANDIDATE_STATUS) FROM CANDIDATES;

         255
         255
         255
3 rows selected
```

Example 3: Using the SIZEOF function for non-textual expressions.

This example uses the personnel sample database.

In the first query, the column BIRTHDAY is defined as DATE VMS which is an an eight-byte datatype. SQL formats the date for display using more than eight characters but the underlying datatype has a storage width of eight bytes. The numeric literal 1 is stored internally using an INTEGER which is four bytes.

In the second query, the column BUDGET_ACTUAL is an INTEGER, a four-byte datatype. Even though all the values of BUDGET_ACTUAL are NULL in the selected rows, SIZEOF still returns the storage width of the column. The numeric literal "99999999999.1" is stored internally as a BIGINT(1), an eight-byte datatype.

```
SQL> SELECT BIRTHDAY, SIZEOF(BIRTHDAY), 1, SIZEOF(1),
cont> FROM EMPLOYEES LIMIT TO 3 ROWS;
 BIRTHDAY
 28-Mar-1947             8   1                       4
 15-May-1954             8   1                       4
 20-Mar-1954             8   1                       4
3 rows selected
SQL> --
SQL> SELECT BUDGET_ACTUAL, SIZEOF(BUDGET_ACTUAL),
cont> 99999999999.1, SIZEOF(99999999999.1)
cont> FROM DEPARTMENTS LIMIT TO 3 ROWS;
 BUDGET_ACTUAL
          NULL             4   99999999999.1                      8
          NULL             4   99999999999.1                      8
          NULL             4   99999999999.1                      8
3 rows selected
```

2.6.2.16 SUBSTRING Function

Substrings return portions of character value expressions. A substring must have the data type CHAR, VARCHAR, LONG VARCHAR, NCHAR, or NCHAR VARYING.

To specify a substring, you must specify the value expression and the FROM keyword, followed by the start position of the value expression. (The first character in the string occupies position 1.) You can optionally add a FOR clause after the FROM clause to specify the length of the value expression after the start position.

The start position and string length values can be a numeric value expression. By default, SQL expects the start position and the string length to be specified in octets. You can use the SET DIALECT or the SET CHARACTER LENGTH statements or the DIALECT or CHARACTER LENGTH clause of the SQL module language header and DECLARE MODULE statement to specify whether the length value is octets or characters.

If you specify a length longer than the string, SQL returns only valid characters in the string and terminates the returned substring after the last valid character.

If either operand of the substring is a null value, the resulting value is also null.

Example: Using SUBSTRING

The following example uses a substring in the WHERE clause of a SELECT statement.

One of the SELECT statement conditions is that 4 characters starting at position 9 must equal the string 'Math', which is extracted using the substring feature.

```
SQL> SELECT * FROM DEGREES
cont> WHERE SUBSTRING(DEGREE_FIELD FROM 9 FOR 4) = 'Math'
cont> AND YEAR_GIVEN > 1980;
 EMPLOYEE_ID   COLLEGE_CODE   YEAR_GIVEN   DEGREE   DEGREE_FIELD
 00167         CALT                 1982   MA       Applied Math
 00168         CALT                 1983   PhD      Applied Math
 00169         MIT                  1981   PhD      Applied Math
 00171         QUIN                 1982   MA       Applied Math
 00176         USCA                 1982   MA       Applied Math
 00212         PRDU                 1983   MA       Applied Math
 00220         DREW                 1982   MA       Applied Math
 00227         PRDU                 1981   MA       Applied Math
 00234         CALT                 1981   PhD      Applied Math
 00242         PRDU                 1982   PhD      Applied Math
```

```
 00243        HVDU                 1981   MA       Applied Math
 00374        STAN                 1982   MA       Applied Math
 00405        MIT                  1982   PhD      Applied Math
 00415        MIT                  1982   PhD      Applied Math
 00418        CALT                 1982   PhD      Applied Math
15 rows selected
```

When you use a substring with the equal (=) conditional operator, the operation is case sensitive.

2.6.2.17 SYSTEM_UID Function

Returns a unique integer that represents the system user.

2.6.2.18 SYSTEM_USER Function

The SYSTEM_USER function returns the user name of the process at the time of the database attach.

If you attach to the database specifying a user name and password in the USER and USING clauses, SQL returns the user name you specify.

The resulting data type is CHAR(31).

2.6.2.19 TRANSLATE Function

SQL provides an alterative TRANSLATE function which uses a comma separated list of arguments.

```
TRANSLATE ( <sourcestring>, <fromstring>, <tostring> )
```

This format of the TRANSLATE function uses two translation character strings to define the translation of specific characters. Any characters in the <sourcestring> which do not appear in the <fromstring> are not replaced. Any characters in the source string which do appear in the <fromstring> are replaced with the character from the corresponding position in the <tostring>. The <fromstring> may be longer than the <tostring> and in this case the matched character is omitted from the result.

If any of the arguments <sourcestring>, <fromstring>, and <tostring> are NULL then the result of the TRANSLATE function is NULL.

The data type of the result is a VARCHAR string with a length and character set equivalent to that of the <sourcestring>.

Examples

Example 1: Eliminating characters

This example removes the single quote punctuation character from names. For example, "O'Hara" will becomes "OHara". This would usually be done during sorting to avoid having the single quote group these names separately, and instead they are ordered between names starting with "Og" (such as Ogdan) and "Oi" (such as Oiler).

```
SQL> select last_name
cont> from EMPLOYEES
cont> order by TRANSLATE (last_name, '"''', '"');
```

Note

In Oracle RDBMS the empty string is considered to be NULL, so an extra `'"'` character was added to the translation strings to avoid a NULL result. This is not required for Oracle Rdb. However, if an ORACLE dialect is used then these Oracle semantics would be possible.

Example 2: Formatting characters

When numeric values are displayed they are normally displayed with leading spaces, however, some applications require leading zeros. This example assumes that the postal code is stored as an INTEGER but needs to report the 5 digits with leading zeros.

```
SQL> select TRANSLATE (CAST(postal_code as CHAR(5)), ' ', '0')
cont> from EMPLOYEES;
```

Example 3: Masking characters

TRANSLATE can be used to mask out characters which should not appear in the output. For instance, when displaying a license number all the letters are required to be converted to 'X' and all digits to '9'.

```
SQL> select TRANSLATE ('2KRW229',
cont>       '01234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ',
cont>       '99999999999XXXXXXXXXXXXXXXXXXXXXXXXXX')
cont> from ...;
 9XXX999
```

Example 4: Transforming text for sorting

Last names often contain special punctuation characters, such as the single quote in O'Sullivan, or D'Amico. Consider this simple example from the PERSONNEL database.

```
SQL> select last_name
cont>  from employees
cont>  where last_name starting with 'D'
cont>  order by last_name;
 LAST_NAME
 D'Amico
 Dallas
 Danzig
 Dement
 Dement
 Dietrich
 Dietrich
7 rows selected
SQL>
```

You can see that the quote punctuation causes the name to sort higher than expected by many applications (such as telephone book listings).

This example removes the single quote punctuation character from names and converts the last name to lowercase so that these names sort within similar names without quote punctuation. However, the original name is displayed as stored in the column.

```
SQL> select last_name
cont>  from employees
cont>  where last_name starting with 'D'
cont>  order by translate (last_name,
cont>                    'ABCDEFGHIJKLMNOPQRSTUVWXYZ''',
cont>                    'abcdefghijklmnopqrstuvwxyz');
 LAST_NAME
 Dallas
 D'Amico
 Danzig
 Dement
 Dement
 Dietrich
 Dietrich
7 rows selected
SQL>
```

The first string argument to TRANSLATE contains the target set of characters, if a character in the name does not match this set it is written to the result unchanged. For example, the trailing spaces are simply copied to the result.

The second string argument contains the translations for those characters. Any upper case characters are transformed into their lower case equivalent. If a single quote character is matched then it is omitted from the result because there was no translation.

Applying these rules will convert "D'Amico" to "damico" so that it sorts between "dallas" and "danzig".

2.6.2.20 TRANSLATE USING Function

The TRANSLATE function translates a character value expression from one character set to another compatible character set.

The characters in the char-value-expr are translated, character-by-character, to the character set indicated by the translation name. If a direct translation exists for a character, it is replaced by the equivalent character in the translation character set. If there is no direct translation for a character, it is replaced by the space character of the translation character set, as shown in the example using the TRANSLATE function.

For example, the Kanji character set contains traditional Kanji characters, Katakana characters, ASCII characters, and Roman characters that are ASCII characters encoded in 2 octets. If a Kanji char-value-expr is translated using the RDB$KATAKANA translation name, those 2-octet Kanji characters that have an equivalent in the 1-octet Katakana character set are translated. The other characters are replaced by the Katakana space character.

Table 2–25 shows the translation name for each character set and to what character set SQL translates particular types of characters.

Table 2–25 Translation Names and Allowable Translations

	char-translation	
Translation Name	**char-value-expression**	**Result**
RDB$DEC_MCS	MCS	MCS
	ASCII	ASCII[1]
RDB$KANJI	Kanji	Kanji
	ASCII	Kanji (Roman characters)
	Katakana	Kanji (Katakana characters)
RDB$DEC_KANJI	Kanji	Kanji
	ASCII	ASCII[1]
	Katakana	Hankaku Katakana
RDB$HANZI	Hanzi	Hanzi
RDB$DEC_HANZI	Hanzi	Hanzi
	ASCII	ASCII[1]
RDB$KOREAN	Korean	Korean
RDB$DEC_KOREAN	Korean	Korean
	ASCII	ASCII[1]
RDB$HANYU	Hanyu	Hanyu
RDB$DEC_SICGCC	Hanyu	Hanyu
	ASCII	ASCII[1]
RDB$DEC_HANYU	Hanyu	Hanyu
	ASCII	ASCII[1]
RDB$KATAKANA	Katakana	Katakana
	Kanji (Katakana)	Katakana
	ASCII	ASCII[1]
RDB$ISOLATINARABIC	Arabic	Arabic
	ASCII	ASCII[1]

[1]Many character sets include ASCII characters. SQL translates the ASCII characters in the source character set to ASCII characters in the target character set.

(continued on next page)

Table 2–25 (Cont.) Translation Names and Allowable Translations

	char-translation	
Translation Name	**char-value-expression**	**Result**
RDB$ISOLATIN1	Extended European Characters ASCII	ASCII[1] Extended European Characters
RDB$ISOLATIN9	Extended European Characters ASCII	ASCII[1] Extended European Characters
RDB$ISOLATINCYRILLIC	Cyrillic	Cyrillic
	ASCII	ASCII[1]
RDB$ISOLATINGREEK	Greek	Greek
	ASCII	ASCII[1]
RDB$ISOLATINHEBREW	Hebrew	Hebrew
	ASCII	ASCII[1]
RDB$DEVANAGARI	Devanagari	Devanagari
	ASCII	ASCII[1]
RDB$SHIFT_JIS	Kanji	Shift_JIS
	ASCII	Shift_JIS (Roman characters)
	Katakana	Shift_JIS (Katakana characters)
RDB$HEX	All Characters	Hexadecimal Equivalent
RDB$UNICODE	All Characters	Unicode
RDB$UTF8	All Characters	UTF8
RDB$WIN_*	same as RDB$ISOLATIN* with same name	
RDB$DOS_LATIN1	same as RDN$ISOLATIN1	
RDN$DOS_LATINUS	same as RDB$ISOLATIN1	
RDB$GB18030	Hanzi	Hanzi
	ASCII	ASCII
RDB$AL24UTFFSS	All Characters	UTF8 Unicode standard 1.1

[1]Many character sets include ASCII characters. SQL translates the ASCII characters in the source character set to ASCII characters in the target character set.

If a character in the source character string is not compatible with the target character set, SQL substitutes a space character for that character.

Example: Using the TRANSLATE function

Use the TRANSLATE function to translate a DEC_MCS column, ENGLISH, to KANJI:

```
SQL> SELECT ENGLISH,FRENCH,JAPANESE FROM COLOURS;
 ENGLISH    FRENCH     JAPANESE
 Yellow     Jaune      黄
 Black      Noir       黒
 Blue       Bleu       青
 Red        Rouge      赤
 White      Blanc      白
 Green      Vert       緑
6 rows selected
SQL> SELECT TRANSLATE (ENGLISH USING RDBVMS$KANJI)
cont> FROM COLOURS;

 Ｂｌａｃｋ
 Ｗｈｉｔｅ
 Ｂｌｕｅ
 Ｒｅｄ
 Ｙｅｌｌｏｗ
 Ｇｒｅｅｎ
6 rows selected
SQL>
```

In the previous example, the TRANSLATE function translates the ASCII characters in the ENGLISH column of the COLOURS table to the Roman characters of the Kanji character set, which uses 2 octets per character. This is useful for concatenation (see Section 2.6.6).

2.6.2.21 TRIM Function

The TRIM function removes either or both leading or trailing spaces, numbers, or characters from any character value expression. SQL returns the specified string minus any leading or trailing characters (or both).

The BOTH option is the default if none is specified. The space character is the default if a string is not specified.

The character value expression that you trim must be defined as data type CHAR, VARCHAR, NCHAR, or NCHAR VARYING. Use the CAST function to convert other data types before using the TRIM function.

SQL returns a run-time error when the trim character is not exactly one character in length.

Examples: Using the TRIM function

Example 1: The following example, though not effective, shows the TRIM function:

```
SQL> SELECT LAST_NAME,
cont> TRIM (LEADING 'H' FROM LAST_NAME)
cont> FROM EMPLOYEES
cont> WHERE LAST_NAME LIKE 'H%';
```

```
 LAST_NAME
 Hall             all
 Harrington       arrington
 Harrison         arrison
 Hastings         astings
 Herbener         erbener
5 rows selected
```

Example 2: Using the TRIM function with the WHERE clause

```
SQL> -- The following INSERT statement helps to show the
SQL> -- TRIM function.
SQL> --
SQL> INSERT INTO EMPLOYEES (LAST_NAME,FIRST_NAME,EMPLOYEE_ID) VALUES
cont> ('  Hillson','Ann','99999');
1 row inserted
SQL> --
SQL> -- If you select columns without specifying the
SQL> -- TRIM function on the WHERE clause, SQL returns only those
SQL> -- last names that start with 'H' and have no leading spaces.
SQL> --
SQL> SELECT LAST_NAME || ', ' || FIRST_NAME
cont> FROM EMPLOYEES
cont> WHERE LAST_NAME LIKE 'H%';
```

```
 Hall          , Lawrence
 Harrington    , Margaret
 Harrison      , Lisa
 Hastings      , Norman
 Herbener      , James
5 rows selected
SQL> --
SQL> -- Add the TRIM function to the WHERE clause to get a complete
SQL> -- list of last names beginning with 'H' including those with
SQL> -- leading spaces.
SQL> --
SQL> SELECT LAST_NAME || ', ' || FIRST_NAME
cont> FROM EMPLOYEES
cont> WHERE TRIM (LEADING ' ' FROM LAST_NAME) LIKE 'H%';
```

```
 Hastings      , Norman
 Harrington    , Margaret
 Hall          , Lawrence
 Harrison      , Lisa
   Hillson     , Ann
 Herbener      , James
6 rows selected
```

Example 3: Using the TRIM function on the SELECT portion of a query in addition to the WHERE clause

```
SQL> -- Add the TRIM function to the SELECT portion of the query
SQL> -- to trim the leading spaces from the display of 'Hillson'.
SQL> -- Note that the LEADING option has been changed to the BOTH
SQL> -- option to trim leading and trailing spaces from the
SQL> -- LAST_NAME column.
SQL> --
SQL> SELECT TRIM (BOTH ' ' FROM LAST_NAME) || ', ' || FIRST_NAME
cont> FROM EMPLOYEES
cont> WHERE TRIM (LEADING ' ' FROM LAST_NAME) LIKE 'H%';

 Hastings, Norman
 Harrington, Margaret
 Hall, Lawrence
 Harrison, Lisa
 Hillson, Ann
 Herbener, James
6 rows selected
```

2.6.2.22 UPPER Function

The UPPER function converts all lowercase characters in a value expression to uppercase characters. This function is useful to maintain consistency in value expressions in the database.

If the result of the value expression is NULL, the function returns a null value.

Example: Using the UPPER function

Use the UPPER function to convert the lowercase characters in DEPARTMENT_NAME from the personnel sample database to uppercase characters:

```
SQL> SELECT DEPARTMENT_NAME, UPPER(DEPARTMENT_NAME)
cont>       FROM DEPARTMENTS
cont>       LIMIT TO 3 ROWS;
 DEPARTMENT_NAME
 Corporate Administration        CORPORATE ADMINISTRATION
 Electronics Engineering         ELECTRONICS ENGINEERING
 Large Systems Engineering       LARGE SYSTEMS ENGINEERING
3 rows selected
```

When you use the UPPER function, SQL follows the rules of the character set of the value expression when converting characters to uppercase. For example, if the character set of the value expression is Hanzi and ASCII, SQL converts only the ASCII characters to uppercase. It does not convert the Hanzi characters.

2.6.2.23 USER Function

The USER function specifies the current active user name for a request and is a synonym for the (CURRENT_USER) function. For definer's rights stored procedures, the returned user name is the definer's user name. For all other requests, it is the current user name of the calling routine or, if no calling routine, the current session user name. The resulting data type is CHAR(31).

Example: Using the USER function

Example 1: Consider an application used by several people to record their sales. The application identifies the sales person by assigning USER to a column in the table:

```
EXEC SQL
INSERT INTO SALES_LOG
        (DATE, AMOUNT, SALES_PERSON)
        VALUES
        (:SALE_DATE, :SALE_AMOUNT, USER)
END-EXEC
```

Example 2: Sales people could then easily retrieve logs of their sales:

```
SQL> SELECT * FROM SALES_LOG
cont>   WHERE SALES_PERSON = USER;
 DATE                          AMOUNT   SALESPERSON
  5-DEC-1988 00:00:00.00          578   FIELDMAN
1 row selected
SQL>
```

2.6.3 Aggregate Functions

Aggregate functions calculate a single value for a collection of rows in a result table. Aggregate functions are sometimes called **statistical functions**.

Table 2–26 describes these functions and the calculated value.

Table 2–26 Aggregate Functions

Function Name	Calculated Value
COUNT	Number of rows in a result table or values in a column
SUM	Sum of a set of values
STDDEV	(standard deviation) The square root of the variance and is expressed in the same units as the source expression.
AVG	Average of a set of values
MAX	Largest value in a set of values
MIN	Smallest value in a set of values
VARIANCE	Statistical measure of variablity from the mean (or average) value.

The following notes generally apply to aggregate functions. An **aggregate function** is a single value derived from one or more sets of values.

- A value expression is used to evaluate a value for each row. The aggregate function operates on these values.
- Null values are not included when SQL evaluates functions. If you specify DISTINCT, redundant values are also not included. If you have set the dialect to SQL99, this null elimination causes a warning to be returned for the SQLCODE or SQLSTATE. See Appendix C for more information on SQLSTATE and SQLCODE.
- If a function has as its argument a value expression that contains a column name that is an outer reference (see Section 2.2.4.2), the value expression cannot include an arithmetic operator. (The only cases where an outer reference makes sense as the argument to a function is in the subquery of a HAVING clause or in a subquery in a select list.)
- You cannot nest functions. This means that a value expression used as an argument to a function cannot include a function.
- The keyword ALL in SUM, AVG, MAX, and MIN has no effect. For instance, specifying MAX (ALL EMPLOYEE_ID) is the same as saying MAX (EMPLOYEE_ID).
- VARIANCE and STDDEV must be passed a single numeric value expression and the result is returned as a DOUBLE PRECISION value. Use the CAST function to alter the result data type.

- VARIANCE, and hence STDDEV, assume that one degree of freedom is used in the calculation of the mean (average) and therefore the divisor is specified as (n-1). For a large number of values in the statistical sample this will not be significant. However, for small samples it may not be desirable, so this default can be changed for the current session by using the SET FLAGS option VARIANCE_DOF(0). Only the values 0 and 1 are allowed.
- The keywords ALL and DISTINCT are not permitted when using the VAR_POP, VAR_SAMP, STDDEV_POP and STDDEV_SAMP statistical functions.

The FILTER clause is provided for all statistical functions. This clause can be used to limit the values included in the COUNT, MAX, MIN, SUM, AVG, STDDEV, and VARIANCE functions.

FILTER can be used to eliminate data from the statistical function so that the generated report can process the data in a single pass. The following example illustrates how the FILTER clause is applied.

```
SQL> select
cont>   max (salary_amount) filter (where salary_end is null),
cont>   max (salary_amount) filter (where salary_end is not null),
cont>   min (distinct salary_amount) filter (where salary_end = salary_start),
cont>   min (distinct salary_amount) filter (where salary_end > salary_start)
cont> from
cont>   salary_history
cont> where
cont>   employee_id = '00164'
cont> group by
cont>   employee_id;

      51712.00        50000.00            NULL        26291.00
1 row selected
SQL>
```

The following sections describe the aggregate functions in more detail.

2.6.3.1 COUNT Function

There are three forms of the COUNT function:

- COUNT (*) calculates the number of rows in a result table. It is the only function that does not allow a specific column name in its argument. The data type of the resulting value expression is an integer.
- COUNT (value-expr) calculates the number of non-NULL values of the value-expr in a result table. The value-expr is evaluated for each row and, if non-NULL, the count is incremented or the value is counted. The data type of the resulting value is an integer.

- COUNT (DISTINCT value-expr) calculates the number of distinct values of the specified value-expr in the result table. The COUNT DISTINCT function eliminates duplicate values from the number it calculates. The value-expr is evaluated for each row and, if non-NULL and if different from previously seen values, the value is counted. It does not count null values in the specified value-expr. The data type of the resulting value is an integer.

If there are no values in the result table to which the COUNT function is applied, the COUNT function returns a zero.

Example: Using the COUNT function

Use the COUNT (*) function to find the number of employees in the personnel database. Use the COUNT (DISTINCT) function to find the number of different states in which they reside:

```
SQL> SELECT COUNT (*) FROM EMPLOYEES;

         100
1 row selected
SQL> SELECT COUNT (DISTINCT STATE) FROM EMPLOYEES;

           3
1 row selected
```

2.6.3.2 SUM Function

The SUM function calculates the total of the values specified by the value expression in its argument. If there are no rows in the result table to which the SUM function is applied, it returns a null value.

The SUM function must refer to a value with a numeric or INTERVAL data type. It returns a value of the same general data type (fixed- or floating-point) big enough to store the result.

If your dialect is set to an ANSI/ISO SQL standard, a warning message is returned if any of the values is NULL.

Example: Using the SUM function

Use the SUM function to calculate the total annual payroll of the company. The SUM function uses all the values that do not have null values in the

column SALARY_AMOUNT within the view CURRENT_SALARY as the result table for its calculation:

```
SQL> SELECT SUM(SALARY_AMOUNT) FROM CURRENT_SALARY;

             3192279.00
1 row selected
```

Because there are no salaries greater than $32,000 to which the SUM function is applied, it returns NULL from the following selection:

```
SQL> SELECT SUM(SALARY_AMOUNT) FROM CURRENT_SALARY
cont> WHERE SALARY_AMOUNT > '32000';

             NULL
1 row selected
```

2.6.3.3 AVG Function

The AVG function calculates the average of the values specified by the value expression in its argument. If there are no rows in the result table to which the AVG function is applied, it returns a null value.

The AVG function must refer to a value with a numeric or INTERVAL data type. In interactive SQL, the value it returns is a floating-point data type for numeric expressions.

In precompiled SQL, SQL module language, and dynamic SQL, the value it returns is rounded off to two decimal places.

The value returned by the AVG function inherits its scale from the data type of the value being averaged. If the value the AVG function refers to has the SMALLINT data type, the result has no scale factor.

If a value is NULL, the row is treated as missing and, if your dialect is set to an ANSI/ISO SQL standard, a warning message is returned.

Example: Using the AVG function

Use the AVG function to find the average salary of all current employees:

```
SQL> SELECT AVG(SALARY_AMOUNT) FROM CURRENT_SALARY;

  3.192279000000000E+004
1 row selected
```

2.6.3.4 MAX Function

The MAX function calculates the largest of the values specified by the value expression in its argument. If there are no values in the result table to which the MAX function is applied, it returns a null value.

The MAX function returns a value of the same data type as the value in its argument for all data types except LIST OF BYTE VARYING.

Example: Using the MAX function

Use the MAX function to find the highest salary paid to an employee:

```
SQL> SELECT MAX(SALARY_AMOUNT) FROM CURRENT_SALARY;

      93340.00
1 row selected
```

2.6.3.5 MIN Function

The MIN function returns the smallest of the values specified by the value expression in its argument. If there are no values in the result table to which the MIN function is applied, it returns a null value.

The MIN function returns a value of the same data type as the column in its argument for all data types except LIST OF BYTE VARYING.

Example: Using the MIN function

Use the MIN function to find the lowest salary paid to an employee:

```
SQL> SELECT MIN(SALARY_AMOUNT) FROM CURRENT_SALARY;

       8687.00
1 row selected
```

2.6.3.6 STDDEV Functions

The STDDEV (standard deviation) function calculates the square root of the variance and is expressed in the same units as the source expression.

Oracle Rdb provides the following statistical functions to standard deviation:

- STDDEV

  This function calculates the standard deviation.

- STDDEV_POP

  This function calculates the standard deviation (the square root of the variance) for the population. It is equivalent to STDDEV with degrees of freedom fixed at 0, that is, SET FLAGS 'VARIANCE_DOF(0)' which is the default setting.

- STDDEV_SAMP

  This function calculates the standard deviation (the square root of the variance) for the subset of sampling of the population. It is equivalent to STDDEV with degrees of freedom fixed at 1, that is, SET FLAGS 'VARIANCE_DOF(1)' which is the default setting. By convention one degree of freedom is used when the sampling of the population is performed.

2.6.3.7 VARIANCE Functions

The VARIANCE function calculates the variablity from the mean (or average) value.

Variance is calculated by the following statistical formula:

$$variance = \frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}[\sum_{i=1}^{n} x_i]^2}{n-1}$$

Where:

xi is one of the elements of x

n is the number of elements in the set x. If n is 1, the variance is defined to be 0.

Oracle Rdb provides the following variance statistical functions:

- VARIANCE

  Calculates the variance of the value set.

- VAR_POP

  This function calculates the variance for the population. It is equivalent to VARIANCE with degrees of freedom fixed at 0, that is, SET FLAGS 'VARIANCE_DOF(0)' which is the default setting.

- VAR_SAMP

  This function calculates the variance for a subset or sampling of the population. It is equivalent to VARIANCE with degrees of freedom fixed at 1, that is, SET FLAGS 'VARIANCE_DOF(1)'. By convention one degree of freedom is used when the sampling of the population is performed.

Example: Using the STDDEV and VARIANCE functions

The following examples show the results from using VARIANCE and STDDEV on a numeric value expression:

```
SQL> SELECT VARIANCE (ALL salary_amount)
cont> FROM salary_history WHERE salary_end IS NULL;

  4.396628811574747E+008
1 row selected
SQL> SELECT VARIANCE (DISTINCT salary_amount),
cont>       COUNT (DISTINCT salary_amount)
cont> FROM salary_history WHERE salary_end IS NULL;

  4.399944817652031E+008           99
1 row selected
SQL> SELECT STDDEV (DISTINCT salary_amount), COUNT (DISTINCT salary_amount)
cont> FROM salary_history WHERE salary_end IS NULL;
```

```
  2.097604542722968E+004           99
1 row selected
SQL> SELECT STDDEV (salary_amount) AS STDDEV
EDIT USING '$$$,$$$,$$9.99'
cont> FROM salary_history WHERE salary_end IS NULL;
         STDDEV
     $20,968.14
1 row selected
```

2.6.4 User-Defined Functions

User-defined functions allow you to execute subprograms written either in 3GL host languages (such as C or SQL Module language) or in the SQL procedural language. There are two classes of user-defined functions:

- The external function feature consists of several discrete pieces: the routine definition, the executable, and the invocation. You define a function that points to the executable form of the routine. You code, compile, and link the routine written in a 3GL language.

  See CREATE ROUTINE Statement and CREATE MODULE Statement for information on creating external routines.

- The SQL function feature consists a module definition containing the function definition. Such SQL functions can include any SQL procedural language features (except those that change the transaction state: SET TRANSACTION, START TRANSACTION, COMMIT and ROLLBACK) and can invoke other functions, call procedures (using the CALL statement) which in turn can activate other user-defined routines.

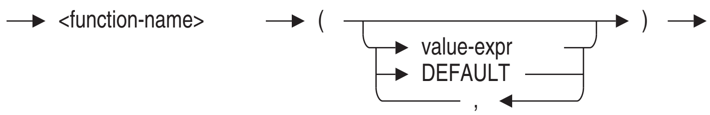
  See CREATE MODULE Statement for information on creating modules and functions in SQL.

The function definitions reside in the database like any other schema object, such as a table or view. Use the SHOW FUNCTION statement to display the names of user-defined functions. See SHOW Statement for more information and examples.

Finally, you refer to the routine within an SQL statement for automatic invocation by the SQL interface.

The following diagram shows how to invoke a user-defined function:

function-invocation =



You invoke a user-defined function from anywhere you can specify a value expression. Some of the locations from which you can invoke an external function are:

- A column using a COMPUTED BY or AUTOMATIC AS value expression clause
- A CHECK clause in a table constraint, column constraint, or domain constraint
- A select expression in a view definition or cursor declaration
- On the right-hand side of a set-assignment-statement of a compound statement or of the SET clause of an UPDATE statement
- A select list or where clause
- A DEFAULT value expression clause for a domain, column or parameter

2.6.5 Database Keys

Database keys (dbkeys) are internal pointers to specific table rows in a database. Application programs can use the DBKEY or ROWID keyword in SQL statements to refer to the database key for a table row. The ROWID keyword is a synonym to the DBKEY keyword. Database key literals are not valid in interactive SQL.

Database keys are considered value expressions. As such, they can be specified as part of a select expression.

SQL statements that retrieve rows by specifying their database keys have the following advantages:

- Fast access: Retrieval through database keys is direct and bypasses any indexed or sequential searches.
- Reduced locking of data: Because access is direct, the database system locks only the row retrieved or updated.

- Uniqueness: Within the database key scope specified in the CREATE DATABASE or DECLARE ALIAS statements, database keys are guaranteed to be unique. This means singleton SELECT statements based on database keys will never return more than a single row, and that they will return the same row, or an error if the row was deleted.

The **scope** of a database key refers to how long the database system guarantees that a particular row's database key will point only to that row and not be used again even if the row is deleted. In ATTACH, CREATE DATABASE, DECLARE ALIAS, and IMPORT statements, you can specify that the database key scope be for the duration of a transaction (the default) or for the duration of an attachment to the database.

Applications that plan to use database keys across transaction boundaries should declare databases with the DBKEY SCOPE IS ATTACH clause, which allows the program to use a database key for a particular table row over the course of many transactions. If another user deletes the row, the database key will not be used again for a newly inserted row, ensuring that the database key is still valid. Any further reference to that DBKEY wukk generate an error.

When you use the DBKEY, some space on the page is not reclaimed until all users are using DBKEY SCOPE IS TRANSACTION and the page is updated. Also, see the RMU RECLAIM command which allows online reclaiming of this space.

Note

Oracle Rdb recommends using DBKEY SCOPE IS TRANSACTION to reclaim space on a database page faster than if you use DBKEY SCOPE IS ATTACH.

The following steps detail how applications use database keys:

1. Declare a database that specifies the DBKEY SCOPE IS ATTACH clause.
2. Start a read-only transaction.
3. Declare a cursor that specifies a result table consisting of the columns and database keys for desired rows.
4. Open the cursor.
5. Fetch the rows of the result table and store them in parameters.
6. Commit the transaction to release locks on database resources.

7. Display a table row stored in parameters, and offer the application user an opportunity to change values in the row.
8. If the application user chooses to change values in the row, copy the row to another set of parameters before allowing the application user to change values. This copy represents the row as originally retrieved from the database.
9. Start a read/write transaction and use a SELECT INTO statement that specifies an authorization identifier to retrieve the row and lock it against changes by other users. Check that other users did not already change the row (since the read-only transaction began) by comparing the row just retrieved with the copy made in the previous step. If other users changed the row, the application might not write over the changed values.
10. If the row in the database matches the row originally retrieved in the read-only transaction, check that the application user made changes by comparing the user's values with the database values.
11. If the application user did change the row, use the database key to specify the row in an UPDATE statement.
12. Commit the transaction.
13. Display another table row stored in parameters and repeat the process.

SQL does not allow the DBKEY keyword in every context that it allows value expressions. Interactive SQL does not allow DBKEY literals. You can use the DBKEY keyword as a value expression only:

- As a select list item (see Section 2.8.1)

  ```
  EXEC SQL DECLARE GET_DBKEYS CURSOR FOR
           SELECT DBKEY FROM EMPLOYEES;
  ```

- In a basic predicate that equates another value expression to the DBKEY keyword

  ```
  EXEC SQL SELECT * FROM EMPLOYEES WHERE DBKEY = :HOST_VAR;
  ```

In addition, the RETURNING DBKEY clause in an INSERT statement directs SQL to return the database key for the row inserted:

```
EXEC SQL INSERT INTO TEMP VALUES (:REAL_VAR)
        RETURNING DBKEY INTO :DBKEY_VAR;
```

Keep the following restrictions in mind when you work with database keys:

- SQL never converts database keys to another data type, but uses the exact value in comparisons or to move to parameters. Oracle Rdb recommends that host language parameters receiving database key values be declared as fixed-length character strings.
- Database keys vary in length. In Oracle Rdb databases, database keys are 8 bytes long for base tables and $8 * number\ of\ tables\ named\ in\ view$ bytes long for views.

  Because of this, you need to declare parameters that are long enough to hold the longest anticipated database key. If a parameter is longer than a database key, SQL truncates parameter values when comparing them to the database keys. If a parameter is not long enough to hold a database key, SQL returns an error when it processes the program or module.

  To determine the length of a database key, retrieve the value of the RDB$DBKEY_LENGTH column from the RDB$RELATIONS system table, as shown in the following example:

  ```
  SQL> SELECT  RDB$DBKEY_LENGTH FROM RDB$RELATIONS
  cont> WHERE RDB$RELATION_NAME = 'CURRENT_JOB';
   RDB$DBKEY_LENGTH
                 16
  1 row selected
  ```

- Rows in result tables or views created by specifying functions, aggregates, the GROUP BY clause, the HAVING clause, or the UNION clause in a select expression do not have database keys.
- Sorting (or any implied sorting for projection) will *not* sort dbkeys in such a way that the dbkeys can be used to retrieve records in sequential order.

  The reason for this behavior is that dbkeys are treated as fixed-length text strings of 8 * n bytes, where n is the number of tables concerned (may be one or more for views). Therefore, sorting dbkeys orders the text bytes according to the default ASCII collating sequence.

2.6.6 String Concatenation Operator

You can use the string concatenation operator (| |) to link two character value expressions. Concatenated value expressions must belong to one of the character data types. In addition, the character sets of the concatenated value expressions must be identical.

Example: Using the Concatenation Operator (| |)

```
SQL> SELECT ENGLISH || JAPANESE
cont> FROM COLOURS;
%SQL-F-INCCSCON, Incompatible character set concatenation between ENGLISH and JA
PANESE
SQL> SELECT TRANSLATE (ENGLISH USING RDBVMS$KANJI) || JAPANESE
cont> FROM COLOURS;

 Ｙｅｌｌｏｗ  黄
 Ｂｌａｃｋ   黒
 Ｂｌｕｅ    青
 Ｒｅｄ     赤
 Ｗｈｉｔｅ   白
 Ｇｒｅｅｎ   緑
6 rows selected
```

The previous example shows that without automatic translation being enabled, incompatible character sets cannot beoncatenated. You must first translate the column to the desired character set. See Section 2.6.2.19 for more information on the TRANSLATE function.

2.6.7 Arithmetic Expressions and Operators

An **arithmetic expression** is a value expression formed by combining one or more numeric value expressions with arithmetic operators. When you use an arithmetic expression in a statement, SQL calculates the numeric value associated with the expression and uses that value when executing the statement.

You cannot use text values in arithmetic expressions whether they are literals, stored in parameters, or stored in table columns.

If either operand of an arithmetic expression is a null value, the resulting value is also null.

The arithmetic operators and their functions are:

```
+       Addition
-       Subtraction
*       Multiplication
/       Division
```

You do not have to use spaces to separate arithmetic operators from value expressions.

You can use parentheses to control the order in which SQL performs arithmetic operations. SQL follows the normal rules of precedence. That is, it evaluates arithmetic expressions in the following order:

1. Value expressions in parentheses

2. Multiplication and division, from left to right
3. Addition and subtraction, from left to right

You can use date-time variables and constants in arithmetic expressions. For details about date-time data types, see Section 2.3.2. Section 2.4.3 provides information about using date-time data types as literals.

The following restrictions apply to date-time arithmetic:

- You cannot use the DATE VMS data type in date arithmetic; you must use the CAST function (CAST(VMS_COL AS TIMESTAMP(2))) or alter the DATE VMS domain to DATE ANSI or TIMESTAMP.
- You must use an interval qualifier with date-time data types in subtraction operations.
- Certain subtraction operations can produce an answer that can be either a YEAR-MONTH interval or a DAY-TIME interval. For example, when subtracting a timestamp from a timestamp or a timestamp from a date, you must specify the qualifier desired as follows:

```
SQL> CREATE TABLE ORDER_TABLE
cont> (PART_NUM INT,
cont> ORDER_LOGGED TIMESTAMP(2),
cont> DELIVERY_DATE TIMESTAMP(2),
cont> TIME_TO_DELIVER COMPUTED BY (DELIVERY_DATE - ORDER_LOGGED) DAY(2)
cont> TO MINUTE, SLOW_DELIVERY COMPUTED BY EXTRACT(DAY FROM
cont> (DELIVERY_DATE - ORDER_LOGGED) DAY(2)) - 30);
```

- You cannot add days to or subtract days from TIME. The result exceeds the allowable range for TIME. The interval day-time column must be a subset of HOURS to SECOND.
- You cannot add hours to or subtract hours from DATE. The interval day-time column must be DAYS only.

The list of valid operations for date-time and interval values appears in Table 2–27.

Table 2–27 Valid Operators Involving Date-Time and Interval Values

Operand 1	Operator	Operand 2	Result Type
Date-time	–	Date-time	Interval
Date-time	+ or –	Interval	Date-time

(continued on next page)

Table 2–27 (Cont.) Valid Operators Involving Date-Time and Interval Values

Operand 1	Operator	Operand 2	Result Type
Interval	+	Date-time	Date-time
Interval	+ or –	Interval	Interval
Interval	* or /	Numeric	Interval
Interval	/	Numeric	Interval
	–	Interval	Interval

Examples: Using Arithmetic Expressions

Example 1: Using an arithmetic expression in a view

An arithmetic expression can be used in a view definition statement. This example defines a view that calculates a payroll deduction for health insurance.

- The select expression in the view definition limits the rows in the view to those for current salary (SALARY_END IS NULL).
- The view columns include the employee's name and a weekly deduction column, calculated using an arithmetic expression from the annual salary for each employee (5% of the weekly salary) as follows:

```
SQL> CREATE VIEW DEDUCT
cont>  (LAST_NAME, FIRST_NAME, AMOUNT)
cont>    AS SELECT
cont>      E.LAST_NAME, E.FIRST_NAME,
cont>      (S.SALARY_AMOUNT / 52) * 0.05
cont>        FROM EMPLOYEES E, SALARY_HISTORY S
cont>        WHERE E.EMPLOYEE_ID = S.EMPLOYEE_ID
cont>        AND
cont>        S.SALARY_END IS NULL;
SQL> SELECT LAST_NAME, FIRST_NAME,
cont> CAST(AMOUNT AS BIGINT(2)) FROM DEDUCT;
 LAST_NAME       FIRST_NAME
 Toliver         Alvin                         49.72
 Smith           Terry                         11.23
 Dietrich        Rick                          17.79
   .
   .
   .
```

Example 2: Using an arithmetic expression in an UPDATE statement

An arithmetic expression can be used to store a value. This example modifies an employee's salary in three steps:

1. The UPDATE statement modifies the row in the SALARY_HISTORY table that represents the employee's old salary by setting the salary-ending date to today's date.
2. The INSERT statement stores a new row using the old EMPLOYEE_ID and SALARY_END date (the one just modified to today's date).
3. The arithmetic expression in the INSERT statement calculates the new salary amount using the old salary (OLD.SALARY_AMOUNT * 1.1) as follows:

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> -- Modify the salary data for employee
SQL> -- with ID 164, adding an ending date:
SQL> --
SQL> UPDATE SALARY_HISTORY
cont>        SET SALARY_END = CAST(CURRENT_DATE AS DATE VMS)
cont>        WHERE
cont>                EMPLOYEE_ID = '00164'
cont>                AND
cont>                SALARY_END IS NULL;
1 row updated

SQL> --
SQL> -- Store a new salary by calculating a 10% raise:
SQL> --
SQL> INSERT INTO SALARY_HISTORY
cont>                (EMPLOYEE_ID, SALARY_START, SALARY_AMOUNT)
cont>        SELECT OLD.EMPLOYEE_ID,
cont>                OLD.SALARY_END,
cont>                (OLD.SALARY_AMOUNT * 1.1)
cont>        FROM SALARY_HISTORY OLD
cont>        WHERE
cont>                OLD.EMPLOYEE_ID = '00164'
cont>                AND
cont>                OLD.SALARY_END = CAST(CURRENT_DATE AS DATE VMS);
1 row inserted
```

```
SQL> --
SQL> -- Check the results.
SQL> --
SQL> SELECT  S.EMPLOYEE_ID,
cont>        S.SALARY_START,
cont>        S.SALARY_END,
cont>        S.SALARY_AMOUNT
cont> FROM    SALARY_HISTORY S
cont> WHERE   S.EMPLOYEE_ID = '00164'
cont> ORDER BY S.SALARY_END DESC ;
 EMPLOYEE_ID   SALARY_START   SALARY_END    SALARY_AMOUNT
 00164          2-Dec-1993    NULL             $56,883.20
 00164         14-Jan-1983     2-Dec-1993      $51,712.00
 00164         21-Sep-1981    14-Jan-1983      $50,000.00
 00164          2-Mar-1981    21-Sep-1981      $26,291.00
 00164          5-Jul-1980     2-Mar-1981      $26,291.00
5 rows selected

SQL> ROLLBACK;
```

Example 3: Using months in a date arithmetic expression

If you add one month to 31 January 1993, the month is incremented as requested, but SQL resets the day to make a valid day of the month. For example, if you enter:

```
SQL> ATTACH 'FILENAME corporate_data';
SQL> SELECT EMPLOYEE_ID, LAST_REVIEW
cont> FROM ADMINISTRATION.PERSONNEL.JOB_HISTORY
cont> WHERE EMPLOYEE_ID = '00164';
 EMPLOYEE_ID   LAST_REVIEW
 00164         NULL
 00164         NULL
2 rows selected
SQL> UPDATE ADMINISTRATION.PERSONNEL.JOB_HISTORY
cont> SET LAST_REVIEW = DATE'1993-01-31' + INTERVAL'1' MONTH
cont> WHERE EMPLOYEE_ID = '00164';
2 rows updated
```

The output is:

```
SQL> SELECT EMPLOYEE_ID, LAST_REVIEW
cont> FROM ADMINISTRATION.PERSONNEL.JOB_HISTORY
cont> WHERE EMPLOYEE_ID = '00164';
 EMPLOYEE_ID   LAST_REVIEW
 00164         1993-02-28
 00164         1993-02-28
2 rows selected
```

2.6.8 Conditional Expressions

A **conditional expression** is an advanced form of the value expression that allows applications to return alternate information within an expression. Table 2–28 describes the conditional expressions in the ANSI/ISO SQL standard that are supported by Oracle Rdb.

Table 2–28 Conditional Expressions

Expression Name	Description
ABS	ABS returns the absolute value of *n*.
NULLIF	NULLIF substitutes NULL when two value expressions are equal; otherwise, returns the first value.
COALESCE	COALESCE returns the first non-NULL value from a series of value expressions; otherwise, returns NULL.
NVL	NVL returns the first non-NULL value from a series of value expressions; otherwise, returns NULL. NVL is a synonym for COALESCE.
NVL2	If the first value expression to NVL2 is not NULL, then return the second value expression; otherwise return the third value expression.
GREATEST	GREATEST returns the greatest non-null value.
LEAST	LEAST returns the least non-null value.
CASE	CASE alters the result of an expression. CASE can also generate or convert null values.
DECODE	Compares expr to srch1 through srchn until a match is found. When a match is found, DECODE returns the corresponding result in resn. If no match is found, DECODE returns the default if specified, null if not.

The following sections describe the SQL implementation of these expressions.

2.6.8.1 ABS Function

The ABS function returns NULL if the passed value expression evaluates to NULL. The datatype of the result is the same as the passed value expression and supports scaled values of these data types: TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE PRECISION, INTERVAL, DECIMAL, NUMERIC and NUMBER.

The absolute value function (ABS) returns NULL if the value expression evaluates to NULL. If the value expression evaluates to a value less than zero then that value is negated so that a positive value is returned. Otherwise the value is returned unchanged. For instance, ABS (-1) will return the value 1.

ABS (*a*) is equivalent to the CASE expression

```
case
    when a < 0 then - a
    else a
end
```

Usage Notes:

- The SQL_FUNCTIONS script still includes the ABS external function definition for those stored definitions (procedures, functions, triggers, views, and so on) or compiled applications that currently use it. However, new references to ABS will use the new builtin conditional expression.
- Applications wishing to continue to use the external function should use delimiters around the ABS function name.

  ```
  SQL> set quoting rules 'SQL92';
  SQL> select "ABS" (v) from T;
  ```

  The delimited name will force the function definition to be used.

Refer to Appendix G for more information on the SQL_FUNCTIONS script.

Example: Using the ABS function on an INTERVAL result of a date subtraction.

```
SQL> select
cont>    ABS ((birthday - current_date) year(3))
cont> from employees
cont> order by employee_id
cont> limit to 10 rows;

  054
  047
  047
  064
  068
  062
  044
  069
  050
  074
10 rows selected
```

Example: Using ABS within a statistical function

```
SQL> -- what is the average time in a job for each employee
SQL> -- exclude anyone on there first job
SQL> select
cont>    employee_id,
cont>    AVG (ABS (EXTRACT (MONTH FROM (job_start - job_end) month (4))))
cont>           as "Average Job" edit using '--,---,--9.99" years"'
cont> from JOB_HISTORY
cont> where employee_id < '00200'
cont> group by employee_id
cont> having COUNT (*) > 1;
 EMPLOYEE_ID          Average Job
 00164                14.00 years
 00165                22.67 years
 00166                20.00 years
 00167                14.50 years
 00168                26.33 years
 00169                22.67 years
...etc...
 00197                26.33 years
 00198                37.00 years
 00199                35.00 years
30 rows selected
%RDB-I-ELIM_NULL, null value eliminated in set function
```

2.6.8.2 COALESCE and NVL Expressions

The COALESCE and NVL expressions return the first non-NULL value from a series of value expressions.

SQL evaluates each value expression in a COALESCE or NVL expression until it can return a non-NULL value. If all the columns specified in the COALESCE or NVL expression contain NULL values, then NULL is returned.

The data type of the resulting expression is a common data type to which all value expressions in the list can be converted. For example, COALESCE(SALARY_AMOUNT, ESTIMATED_BONUS, 1.23E+5) results in a DOUBLE PRECISION result because at least one argument is a floating point value.

The following example replaces the stored NULL value in the MIDDLE_ INITIAL column of the EMPLOYEES table with a hyphen:

```
SQL> SELECT FIRST_NAME, LAST_NAME, MIDDLE_INITIAL,
cont> COALESCE(MIDDLE_INITIAL, '-')
cont> FROM EMPLOYEES
cont> WHERE LAST_NAME LIKE 'L%';
 FIRST_NAME   LAST_NAME        MIDDLE_INITIAL
 Jo Ann       Lapointe         C                 C
 Hope         Lapointe         NULL              -
 Stan         Lasch            P                 P
 Norman       Lasch            NULL              -
 Peter        Lengyel          A                 A
 Peter        Lonergan         V                 V
6 rows selected
```

2.6.8.3 CASE Expressions

There are many situations where you might find it useful to alter the result of an expression. For example, you might have a table column called WORK_STATUS containing the data 0, 1, and 2 meaning Inactive, Full time, and Part time, respectively. The single character is more efficient to store than the definition of the character in the database. However, the definition of the single character is not always intuitive.

There may also be times when you want to generate null values based on the information derived from the database or, conversely, convert a null value into a more concrete value like zero (0). The CASE expressions provide an easy solution to these problems.

There are two types of CASE expressions:

- Simple—matches two value expressions for equality
- Searched—allows complex predicate, including subqueries

An example of the simple case expression follows:

```
SQL> SELECT LAST_NAME, FIRST_NAME,
cont>   CASE STATUS_CODE
cont>     WHEN '1' THEN 'Full time'
cont>     WHEN '2' THEN 'Part time'
cont>     WHEN NULL THEN 'Unknown'
cont>     ELSE 'Inactive'
cont>   END
cont> FROM EMPLOYEES;
 LAST_NAME        FIRST_NAME
 Smith            Terry        Part time
 O'Sullivan       Rick         Full time
   .
   .
   .
 Sarkisian        Dean         Part time
 Stornelli        James        Full time
 Hall             Lawrence     Full time
 Mistretta        Kathleen     Full time
 James            Eric         Inactive
 MacDonald        Johanna      Full time
 Dement           Alvin        Full time
 Blount           Peter        Full time
 Herbener         James        Full time
 Ames             Louie        Full time
100 rows selected
```

When SQL encounters the first WHEN clause that matches the primary value expression following the CASE keyword, it evaluates the THEN clause. If no matching values are found, the ELSE clause is evaluated. If the ELSE clause is missing, NULL is the returned value. For example:

```
SQL> SELECT PRODUCT_NAME,
cont>   CASE
cont>      WHEN QUANTITY <= 0 THEN 'On back order'
cont>      WHEN QUANTITY > 0 THEN
cont>         CAST(QUANTITY AS VARCHAR(10)) || ' in stock'
cont>   END
cont> FROM INVENTORY;
 PRODUCT_NAME
 Staples-boxes     20 in stock
 Staplers-each     3 in stock
 Tape-rolls        On back order
 Calendars-each    25 in stock
 Tape disp.-each   On back order
 Desk cleaner      NULL
6 rows selected
```

An example of the searched case expression follows:

```
SQL> SELECT PRODUCT_NAME,
cont>   CASE
cont>      WHEN QUANTITY <= 0 THEN 'On back order'
cont>      WHEN QUANTITY > 0 THEN
cont>         CAST(QUANTITY AS VARCHAR(10)) || ' in stock'
cont>      ELSE                                          -- must be NULL
cont>        'New Item - awaiting stock'
cont>   END
cont> FROM INVENTORY;
 PRODUCT_NAME
 Staples-boxes     20 in stock
 Staplers-each     3 in stock
 Tape-rolls        On back order
 Calendars-each    25 in stock
 Tape disp.-each   On back order
 Desk cleaner      New Item - awaiting stock
6 rows selected
```

The searched case expression allows arbitrary expressions in each WHEN clause, as shown in the previous example. The simple case expression is a shorthand method of specifying the searched case expression.

For the simple and searched case expressions, the data types of the value expressions of the WHEN clause must be comparable, and the data types of the value expressions of the THEN clause must be comparable.

All subqueries in a CASE expression are evaluated. It is the results of these subqueries that are conditionalized by the CASE expression and not the actual evaluation.

If any subquery (which must return at most a single row and column) returns more than one row, the following exception is generated:

```
%RDB-E-MULTIPLE_MATCH, record selection criteria should identify only one
record; more than one record found
```

A workaround is to add one of the following clauses to the subquery:

- LIMIT TO 1 ROW

  This ensures that only one row is returned. For example:

```
   .
   .
   .
cont> WHEN A IS NOT NULL
cont> THEN (SELECT A FROM T WHERE B = Y
cont>       LIMIT TO 1 ROW)
   .
   .
   .
```

The WHEN condition ignores this row if it is not valid.

- Duplicate the WHEN clause Boolean inside the subquery predicate

  For example:

```
SQL> --
SQL> -- Change the following syntax from
SQL> --
   .
   .
   .
cont> WHEN A IS NOT NULL
cont> THEN (SELECT A FROM T WHERE B = Y)
   .
   .
   .
SQL> --
SQL> -- to include the Boolean inside the subquery
SQL> --
   .
   .
   .
cont> WHEN A IS NOT NULL
cont> THEN (SELECT A FROM T WHERE B = Y AND A IS NOT NULL)
   .
   .
   .
```

  In this example, when the WHEN clause evaluates as FALSE, so will the WHERE predicate from the subquery and, therefore, will return no rows.

In either of the above cases, the correct results are returned from the query.

2.6.8.4 GREATEST and LEAST Functions

The GREATEST and LEAST functions accept a list of two or more value expressions (all of which must be of comparable types) and return the greatest value from the list for the GREATEST function and the least value from the list for the LEAST function. The value expressions specified can be column references, subselects, function calls, literal values, and other complex value expressions.

The data type of the resulting expression is a common data type to which all value expressions in the list can be converted. For example, LEAST(10, 10.3, 123E100) results in a DOUBLE PRECISION result because at least one literal is DOUBLE PRECISION.

If the result data type resolves to a fixed CHARACTER string, then GREATEST and LEAST return a CHARACTER VARYING (also known as VARCHAR) string with the maximum length.

The NULL keyword can appear in the list but is ignored. However, not all value expressions can be specified as NULL. That is, a non-NULL value expression must be in the list so that the data type for the expression can be determined.

The GREATEST and LEAST functions can result in NULL only if at run time all value expressions result in NULL.

If DATE VMS literals are used as an argument to the GREATEST or LEAST function, the date string must be prefixed with the type DATE VMS so that SQL will accept it as a DATE VMS literal and not a string literal. See the following examples.

Examples: Using the GREATEST and LEAST functions

Example 1 Using CHARACTER Versus DATE VMS Literals

```
SQL> -- Different results are returned by the LEAST function (and
SQL> -- the GREATEST function) if the parameters are treated as
SQL> -- CHARACTER or DATE VMS literals.  This is because it is
SQL> -- the data types of the source expressions that determine the
SQL> -- the result data type of the LEAST (and GREATEST) functions.
SQL> select LEAST ('1-APR-2001', '10-JAN-2000'),
cont> LEAST (DATE VMS'1-APR-2001', DATE VMS'10-JAN-2000')
cont> from rdb$database;

 1-APR-2001    10-JAN-2000 00:00:00.00
1 row selected
```

Example 2 finds the least value for the birthday of employees from two employees tables.

Example 2 Using the LEAST Function

```
SQL> SELECT LEAST (M.BIRTHDAY, E.BIRTHDAY, :SUPPLIED_DATE)
cont> FROM EMPLOYEES E, JOB_HISTORY JH, DEPARTMENTS D, EMPLOYEES M
cont> WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID AND
cont> ...
```

2.6.8.5 NULLIF Expressions

The NULLIF expression is used to substitute NULL when two value expressions are equal. For example, if the data stored in column ADDRESS_DATA_1 or ADDRESS_DATA_2 are space characters, the NULLIF expression replaces the space value with the NULL value.

```
SQL> BEGIN
cont>   INSERT INTO EMPLOYEES
cont>     (EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
cont>      ADDRESS_DATA_1, ADDRESS_DATA_2)
cont>   VALUES
cont>     (:EMP_ID, 'Clinton', 'William',
cont>      NULLIF(:ADD_1, ' '),
cont>      NULLIF(:ADD_2, ' '));
cont> END;
SQL>
SQL> SELECT LAST_NAME, ADDRESS_DATA_1, ADDRESS_DATA_2
cont> FROM EMPLOYEES
cont> WHERE EMPLOYEE_ID = :EMP_ID;
 LAST_NAME       ADDRESS_DATA_1              ADDRESS_DATA_2
 Clinton         NULL                        NULL
1 row selected
```

The following example substitutes NULL when the MIDDLE_INITIAL column of the EMPLOYEES table contains space characters:

```
SQL> SELECT LAST_NAME,
cont>   NULLIF (MIDDLE_INITIAL, ' '),
cont>   FIRST_NAME
cont> FROM EMPLOYEES
cont> WHERE EMPLOYEE_ID IN ('00191', '00198');
 LAST_NAME              FIRST_NAME
 Pfeiffer        I      Karen
 Gehr            NULL   Leslie
2 rows selected
```

2.6.8.6 NVL2 Expressions

NVL2 lets you compute a value based on whether a specified expression is null or not null. If the first value expression is not null then the second value expression is returned as the function result. Otherwise, the final value expression is returned. The data type function is derived as a common data type of the second and third value expressions.

For example, when the JOB_END date in JOB_HISTORY is NULL then that indicates the current job for that employee. The following example uses NVL2 to annotate the output from a query on JOB_HISTORY displaying either "current job" or "prior job" based on the NULL attribute of the JOB_END column.

```
SQL> select employee_id, job_start, job_end,
cont> NVL2 (job_end, 'prior job', 'current job')
cont> from job_history
cont> where employee_id < '00180'
cont> order by employee_id, job_start;
 EMPLOYEE_ID   JOB_START     JOB_END
 00164          5-Jul-1980   20-Sep-1981   prior job
 00164         21-Sep-1981   NULL          current job
 00165          1-Jul-1975    4-Sep-1977   prior job
 00165          5-Sep-1977    7-Apr-1979   prior job
 00165          8-Apr-1979    7-Mar-1981   prior job
 00165          8-Mar-1981   NULL          current job
    .
    .
    .
```

The following example shows whether the income of some employees is made up of SALARY plus COMMISSION, or just SALARY, depending on whether the COMMISSION_PCT column of EMPLOYEES is null or not.

```
SQL> SELECT last_name, salary_amount,
cont>       NVL2 (commission_pct,
cont>             salary_amount + (salary_amount * commission_pct),
cont>             salary_amount) as Income edit using SALARY
cont> FROM employees e, salary_history sh
cont> WHERE last_name like 'B%'
cont>   and e.employee_id = sh.employee_id
cont>   and salary_end is null
cont> ORDER BY last_name;
 E.LAST_NAME      SH.SALARY_AMOUNT        INCOME
 Babbin                 $20,150.00    $20,956.00
 Bartlett               $14,817.00    $15,261.51
 Bartlett               $38,223.00    $38,987.46
 Belliveau              $54,649.00    $55,741.98
 Blount                 $63,080.00    $64,341.60
 Boyd                   $30,275.00    $30,275.00
 Boyd                   $24,166.00    $24,166.00
 Brown                  $50,357.00    $50,357.00
 Burton                 $23,053.00    $23,053.00
9 rows selected
SQL>
```

2.7 Predicates

A **predicate** specifies a condition that SQL evaluates as true, false, or unknown. Predicates are also called conditional expressions. You can specify several different types of predicates with different conditional operators. The different types of predicates are:

- Basic

- BETWEEN
- Complex
- CONTAINING
- EXISTS
- IN
- IS NULL
- LIKE
- Quantified
- SINGLE
- STARTING WITH
- UNIQUE

When you compare character value expressions, if automatic translation has not been enabled, character sets of those value expressions must be identical.

Some predicates have a specific behavior when used with the DEC Multinational Character Set (MCS). This behavior is described in the following sections.

The following list describes multinational character set behavior that applies to predicates:

- The character *ñ* is always treated as different from the character *n,* in keeping with the practices of the Spanish language. In a similar manner, the character *ç* is treated the same as the character *c*, in keeping with the practices of the French language.
- The character *ü* is treated the same as the character *u* for many languages, but is sorted between the characters *x* and *z* (with the *y*s) for Danish, Norwegian, and Finnish languages.

The following diagram shows the syntax for predicates:

predicate =

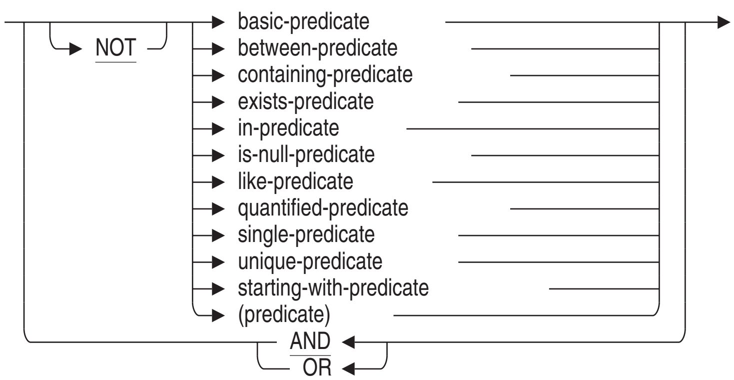


Table 2–29 summarizes how SQL evaluates the different conditional operators.

Table 2–29 SQL Conditional Operators

Comparison Operator	Predicate Is:
=	True if the two value expressions are equal.
<>	True if the two value expressions are not equal.
^=	True if the two value expressions are not equal.
!=	True if the two value expressions are not equal. This basic predicate is only available if you set the ORACLE LEVEL1 or ORACLE LEVEL2 dialects.
<	True if the first value expression is less than the second value expression.
<=	True if the first value expression is less than or equal to the second value expression.
>	True if the first value expression is greater than the second value expression.
>=	True if the first value expression is greater than or equal to the second value expression.

(continued on next page)

Table 2–29 (Cont.) SQL Conditional Operators

Comparison Operator	Predicate Is:
ALL	True if the specified relationship is true for every row (which must be only a single column wide) of the result table specified by the column select expression. Also true if the result table is empty. ALL is a type of quantified predicate.
ANY (SOME)	True if the specified relationship is true for at least one row (which must be only a single column wide) of the result table specified by the column select expression. ANY is a type of quantified predicate. (SOME is the same as ANY. The keywords are synonymous.)
BETWEEN	True if the first value expression is greater than the second value expression and less than the third value expression, or equal to one of them.
NOT BETWEEN	True if the first value expression is not greater than the second value expression and less than the third value expression, and not equal to either of them.
CONTAINING	True if the string specified by the second value expression is found within the string specified by the first. Not case sensitive.
NOT CONTAINING	True if the string specified by the second value expression is not found within the string specified by the first. Not case sensitive.
EXISTS	True only if the number of rows in the result table specified by the column select expression is not zero.
NOT EXISTS	True only if the number of rows in the result table specified by the column select expression is not zero.
IN	True if the value expression on the left is equal to one of the values specified by the list of value expressions (including column select expressions) on the right.
NOT IN	True if the value expression on the left is not equal to any of the values specified by the list of value expressions or column select expressions on the right.
IS NULL	True if the value expression is null.
IS NOT NULL	True if the value expression is not null.

(continued on next page)

Table 2–29 (Cont.) SQL Conditional Operators

Comparison Operator	Predicate Is:
LIKE	True if the first expression matches the pattern in the second value expression. LIKE uses these special characters: % (percent sign) Matches any string _ (underscore) Matches any single character
NOT LIKE	True if the first expression does not match the pattern in the second value expression.
SINGLE	True if the result table specified by the column select expression includes exactly one row.
NOT SINGLE	True if the result table specified by the column select expression includes more than one row or zero rows.
STARTING WITH	True if the first characters of the first value expression match those specified in the second value expression. Case sensitive.
NOT STARTING WITH	True if the first characters of the first value expression do not match those specified in the second value expression. Case sensitive.
UNIQUE	True if no duplicate rows exist in the result table of a column select expression.
NOT UNIQUE	True if no duplicate rows exist in the result table of a column select expression.

Note

Except for the IS NULL, EXISTS, and SINGLE operators, if either operand in a predicate is null, the value of the predicate is unknown.

You cannot use a value of the LIST OF BYTE VARYING data type for either operand in a comparison predicate. For more information, see Section 2.3.7.

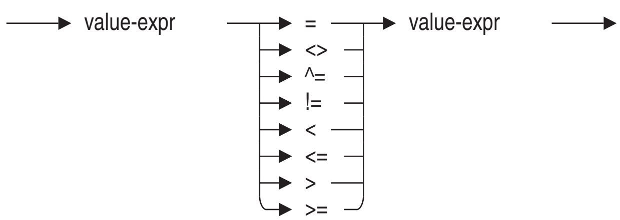
When you use the DEC_MCS or ASCII character set, SQL compares character string literals according to the ASCII collating sequence. Therefore, it considers lowercase letters to have a greater value than uppercase letters, and considers the letters near the beginning of the alphabet to have a lesser value than those near the end.

```
'a' > 'A'
'a' > 'Z'
'a' < 'z'
'A' < 'z'
'A' < 'Z'
```

2.7.1 Basic Predicate

A basic predicate compares two values.

basic-predicate =



See Section 2.6 for details on value expressions.

Example: Using a basic predicate in a SELECT statement

The following SELECT statement uses a basic predicate that contains a column select expression to find employees who make a higher-than-average salary:

```
SQL> SELECT DISTINCT EMPLOYEE_ID FROM SALARY_HISTORY
cont>       WHERE SALARY_AMOUNT >
cont>               (SELECT AVG(SALARY_AMOUNT)
cont>                FROM SALARY_HISTORY);
 EMPLOYEE_ID
 00164
 00168
   .
   .
   .
```

In this example, the predicate is:

```
SALARY AMOUNT > (SELECT AVG(SALARY_AMOUNT) FROM SALARY_HISTORY)
```

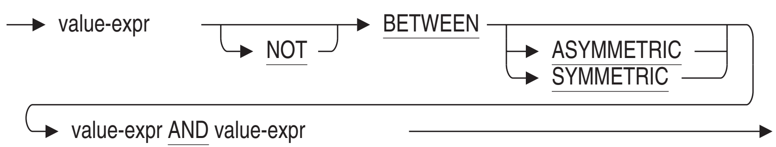
In addition to the <> basic predicate, the ^= and != are available for inequality comparisons. However, != is only available if you set the ORACLE LEVEL1 dialect. [1] See SET DIALECT Statement for information on setting dialects.

[1] Since in other dialects ! is considered a comment character.

2.7.2 BETWEEN Predicate

A BETWEEN predicate compares a value with a range of values.

between-predicate =



See Section 2.6 for details on value expressions.

ASYMMETRIC is the default.

The BETWEEN predicate is a simpler way of representing conditions that can be represented using other conditional operators:

```
value1 BETWEEN value2 AND value3
```

Using the BETWEEN predicate is the same as using the following complex predicate:

```
value1 >= value2
AND
value1 <= value3
```

ASYMMETRIC

By default, the BETWEEN predicate depends on the ordering of the values. i.e. the first value expression needed to be less than or equal to the second value expression and was evaluated as equivalent to: $V0 \geq V1$ and $V0 \leq V2$. This is demonstrated by the following example.

```
SQL> select a from t where a between asymmetric 2 and 4;
           A
           2
           3
           4
3 rows selected
```

The following query returns zero matches because the value expressions are out of order.

```
SQL> select a from t where a between asymmetric 4 and 2;
0 rows selected
```

SYMMETRIC

This alternate format for BETWEEN allows simpler comparision of unordered value expressions, as can be seen in these examples which return the same results. This comparision is equivalent to: ($V0 \geq V1$ and $V0 \leq V2$) or ($V0 \geq V2$ and $V0 \leq V1$).

For example:

```
SQL> select a from t where a between symmetric 2 and 4;
           A
           2
           3
           4
3 rows selected
SQL> select a from t where a between symmetric 4 and 2;
           A
           2
           3
           4
3 rows selected
```

Note that NOT BETWEEN operation also changes when using SYMMETRIC

This first query using ASYMMETRIC returns all values not in the specified range.

```
SQL> select a from t where a not between asymmetric 2 and 4;
           A
           1
           5
2 rows selected
```

In this next query the range values is out of order and the BETWEEN predicate returns an empty set of matches, and therefore NOT BETWEEN returns all rows in the example table.

```
SQL> select a from t where a not between asymmetric 4 and 2;
           A
           1
           2
           3
           4
           5
5 rows selected
```

Contrast this to SYMMETRIC which returns the same set of values for either ordering of values:

```
SQL> select a from t where a not between symmetric 2 and 4;
          A
          1
          5
2 rows selected
SQL> select a from t where a not between symmetric 4 and 2;
          A
          1
          5
2 rows selected
```

Example: Using the BETWEEN predicate with character columns

The following example uses a BETWEEN predicate to find the names of employees whose names begin with the character *B*:

```
SQL> SELECT LAST_NAME
cont> FROM EMPLOYEES
cont> WHERE LAST_NAME
cont> BETWEEN 'B' AND 'C';
 LAST_NAME
 Babbin
 Bartlett
 Bartlett
 Belliveau
 Blount
 Boyd
 Boyd
 Brown
 Burton
9 rows selected
```

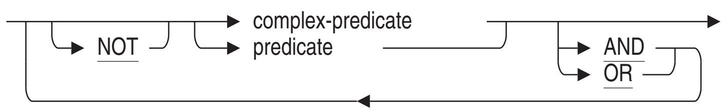
This example can retrieve more names than those of employees whose last names begin with the character *B*. An employee whose last name was *C* would be included in the result. To omit that employee, use the following BETWEEN predicate:

```
 BETWEEN 'B' AND
'Bzzzzzz'.
```

2.7.3 Complex Predicate

A **complex predicate** combines any number of predicates with the Boolean operators AND, OR, and NOT. Boolean operators are also called logical operators.

complex-predicate =



When nesting predicates, you must enclose them in parentheses. SQL evaluates parts of a complex predicate in this order:

1. Predicates enclosed in parentheses

   If there are nested predicates in parentheses, the innermost predicate is evaluated first.

2. Predicates preceded by NOT
3. Predicates combined with AND
4. Predicates combined with OR

Table 2–30, Table 2–31, and Table 2–32 summarize how SQL evaluates predicates combined with Boolean operators. Such tables are often called **truth tables.**

Table 2–30 Boolean Operator: AND

A	B	A AND B
True	False	False
True	True	True
False	False	False
False	True	False
True	Unknown	Unknown
False	Unknown	False
Unknown	True	Unknown
Unknown	False	False
Unknown	Unknown	Unknown

Table 2–31 Boolean Operator: OR

A	B	A OR B
True	False	True
True	True	True
False	False	False
False	True	True
True	Unknown	True
False	Unknown	Unknown
Unknown	True	True
Unknown	False	Unknown
Unknown	Unknown	Unknown

Table 2–32 Boolean Operator: NOT

A	NOT A
True	False
False	True
Unknown	Unknown

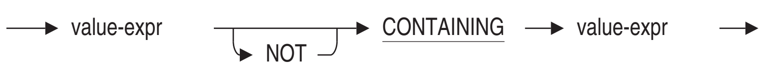
Note

The fact that NOT A is evaluated as unknown when A is unknown can be confusing in queries that refer to tables with null values. It means that a NOT predicate is not necessarily evaluated as true for all rows of a column for which the same predicate without NOT is evaluated as false. In other words, the result of a query that contains NOT A is not necessarily the complement of the result of the same query that contains only A.

2.7.4 CONTAINING Predicate

A CONTAINING predicate tests whether or not the string expression specified in the second value expression is contained within the string expression specified by the first.

containing-predicate =



The CONTAINING predicate is not case sensitive.

The CONTAINING predicate is sensitive to diacritical markings used in the DEC Multinational Character Set. Therfore, *a* matches *A*, but neither matches *á*, *à*, *ä*, *Á*, *À*, *Â* and so on.

In Spanish, *ch* and *ll* are treated as if they were unique single characters.

If you use a collating sequence, the CONTAINING predicate will not be sensitive to diacritical markings used in the DEC Multinational Character Set.

Example: Using the CONTAINING predicate

```
SQL> -- Note that CONTAINING is not case sensitive.
SQL> -- Although 'TOL' is typed in all uppercase letters,
SQL> -- SQL still returns Toliver, which is
SQL> -- in uppercase and lowercase letters.
SQL> --
SQL> SELECT E.LAST_NAME FROM EMPLOYEES E WHERE
cont> E.LAST_NAME CONTAINING 'TOL';
 LAST_NAME
 Toliver
1 row selected
```

2.7.5 EXISTS Predicate

An EXISTS predicate tests whether or not the result table specified in a column select expression is empty.

exists-predicate =

```
──► EXISTS ( select-expr )          ──────►
```

If the result table specified in the select expression has one or more rows, SQL evaluates the EXISTS predicate as true. Otherwise, the predicate is false. An EXISTS predicate cannot be unknown.

Because it only checks for the existence of rows, an EXISTS predicate does not require that the result table from its column select expression be a single column wide (see Section 2.8.2 for details on column select expressions). For EXISTS predicates, an asterisk (*) wildcard in the column select expression can refer to a multicolumn table (see the following example).

Example: Using the EXISTS predicate

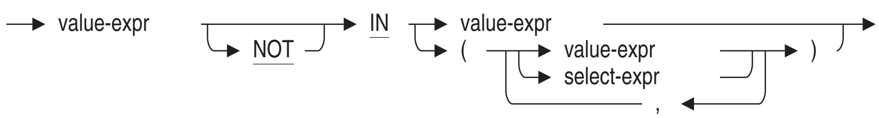
The following example illustrates the EXISTS predicate. It parallels Example 2 in Section 2.7.9, which uses the = ANY predicate to find employees with college degrees, and the NOT (= ANY) predicate to find the names of employees who do not have college degrees.

```
SQL> SELECT E.LAST_NAME, E.EMPLOYEE_ID
cont> FROM      EMPLOYEES E
cont> WHERE     EXISTS
cont>           -- Notice that the column select expression uses a wildcard,
cont>           -- which is valid for multicolumn tables only in EXISTS
cont>           -- predicates:
cont>           (SELECT *
cont>             FROM DEGREES D
cont>             WHERE D.EMPLOYEE_ID =
cont>                   E.EMPLOYEE_ID);
 LAST_NAME        EMPLOYEE_ID
 Toliver          00164
 Smith            00165
 Dietrich         00166
    .               .
    .               .
    .               .
 Blount           00418
 MacDonald        00435
 Herbener         00471
99 rows selected
```

2.7.6 IN Predicate

An IN predicate compares a value with another value or a collection of values.

in-predicate =



See Section 2.6 for details on value expressions. See Section 2.8.2 for details on column select expressions.

All forms of the IN predicates can be represented using other conditional operators.

- `value-expr IN value-expr`

  is the same as

  ```
  value-expr IN (value-expr)
  ```

  which is the same as the basic predicate

  ```
  value-expr = value-expr
  ```

  (as long as the value expression on the right is not a host structure that expands to more than one parameter)

- `value-expr IN (value-expr1, value-expr2, value-expr3)`

  is the same as the complex predicate

  ```
  value-expr = value-expr1
  OR
  value-expr = value-expr2
  OR
  value-expr = value-expr3
  ```

  (in this case, any of the value expressions on the right can be a host structure that expands to more than one parameter)

- `value-expr IN (col-select-expr1, val-expr2, col-select-expr3)`

  is the same as the quantified predicate

  ```
  value-expr = ANY (col-select-expr1)
  OR
  value-expr = val-expr2
  OR
  value-expr = ANY (col-select-expr3)
  ```

(in this case, any of the value expressions on the right can be a host structure that expands to more than one parameter)

Example: Using the IN predicate with a value expression list

The following example uses an IN predicate with a list of value expressions (in this case, string literals) to find the number of employees who live in New England:

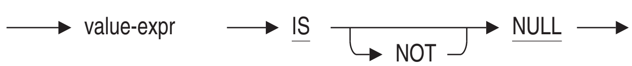
```
SQL> SELECT    COUNT(*)
cont> FROM      EMPLOYEES
cont> WHERE     STATE IN
cont>           ('CT', 'RI', 'MA', 'VT', 'NH', 'ME');

        100
1 row selected
```

2.7.7 IS NULL Predicate

An IS NULL predicate tests for null values in value expressions.

is-null-predicate =



See Section 2.6 for details on value expressions.

SQL never evaluates an IS NULL predicate as unknown; it is always true or false. If the value expression is null, SQL evaluates the predicate as true. If the value expression is not null, the predicate is false.

Use an IS NULL predicate to retrieve rows with null values in particular columns. An IS NULL predicate is the only way to construct a query that includes rows in a result table by testing whether or not particular columns in the rows have null values. Other constructions such as NOT LIKE or <> (not equal) do not include rows with null values in their result tables.

Example: Retrieving rows based on null values with the IS NULL predicate

The following example illustrates that you must use IS NULL predicates to retrieve rows with null values:

```
SQL> -- The following query does not include rows that
SQL> -- have null values in the MIDDLE_INITIAL column:
SQL> --
SQL> SELECT COUNT(*) FROM EMPLOYEES
cont> WHERE NOT (MIDDLE_INITIAL = 'V');

         60
1 row selected
```

```
SQL> --
SQL> -- To get a count of rows that have no values stored in
SQL> -- the MIDDLE_INITIAL column, use an IS NULL predicate.
SQL> --
SQL> SELECT COUNT(*) FROM EMPLOYEES
cont> WHERE MIDDLE_INITIAL IS NULL;

          36
1 row selected
```

2.7.8 LIKE Predicate

A LIKE predicate searches character string literals for pattern matches. The LIKE predicate is case sensitive; it considers uppercase and lowercase forms of the same character to be different characters.

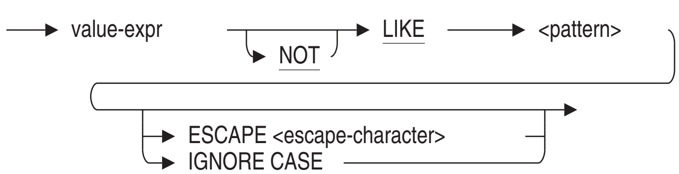
Because the LIKE predicate is case sensitive, searches for uppercase characters do not include lowercase characters in the DEC Multinational Character Set. The reverse is also true. For example, `LIKE "Ç"` will retrieve a different set of records than `LIKE "ç"`.

The LIKE predicate is sensitive to diacritical markings used in the DEC Multinational Character Set. Therfore, *a* matches *A*, but neither matches *á*, *à*, *ä*, *Á*, *À*, *Â* and so on.

In Spanish, *ch* and *ll* are treated as if they are unique single letters. For example, if a domain is defined with the collating sequence SPANISH, then `LIKE "c%"` will not retrieve the word *char* but will retrieve the word *cat.*

The LIKE predicate has this form:

like-predicate =



pattern =

escape-character =

```
───► char-value-expr ──────►
```

SQL interprets the value-expr argument as a character string and compares it to the pattern. The pattern must be a value expression with a text data type.

Within the pattern, the percent sign (%), underscore (_), and escape characters have special meaning.

- The percent sign represents any string of characters, including no characters at all. The percent sign is a wildcard character.
- The underscore represents any single character.
- An escape character causes SQL to interpret a wildcard character as itself to search for character strings containing the wildcard character. The value of the escape character must be 1 character in length.

  Table 2–33 explains the valid sequences allowed for escape characters.

Table 2–33 Escape Character Sequences

Character in Pattern	Character Matched
%	any string
_	any character
escape-character %	%
escape-character _	_
escape-character escape-character	escape-character

  You can only specify the percent sign, underscore, or the escape-character itself. Any other character is invalid and an error is returned.

All other characters represent themselves.

Note

If you do not need to use wildcards, Oracle Rdb recommends that you use the basic predicate instead of the LIKE predicate. If you need to use wildcards, always use the percent sign.

Oracle Rdb can improve the performance of certain SQL queries that include LIKE predicates that do not contain IGNORE CASE clauses. This type of query optimization occurs when the LIKE operator string begins with a pattern of one or more characters that does not include the wildcard character (% or _) or the escape character.

For example, Oracle Rdb. can optimize the following LIKE predicate because the LIKE predicate string begins with the pattern “RAN”, which does not include the wildcard character or the escape character and contains no IGNORE CASE clause:

```
SELECT * FROM EMPLOYEES WHERE LAST_NAME LIKE 'RAN%D%';
```

When the prefix of the pattern is known, namely “RAN”, then Oracle Rdb. uses that prefix to establish an index range to improve query performance. The pattern can be any arbitrary expression, and does not need to be a compile-time constant.

In contrast, Oracle Rdb does not apply the index range optimization to the LIKE predicate of the following query because the pattern begins with the wildcard character which prevents efficient index retrieval:

```
SELECT * FROM EMPLOYEES WHERE LAST_NAME LIKE '%RAN';
```

The LIKE predicate has the following restrictions:

- The LIKE predicate does not pad its argument (pattern) with blank spaces for comparison with value expressions that are not the same length as the argument. This means that the LIKE predicate does not find matches for some patterns when you might expect it to find matches.

  For example, the CHAR data type is a fixed-length string. When you insert data into a CHAR column and the data has fewer characters than the column definition, the remainder of the string literal is padded with blank spaces. In contrast, a VARCHAR data type is a varying-length string. The inserted string literal is not padded with blank spaces. Because the LIKE predicate does a character-for-character comparison, the value in a CHAR data type column, which is padded with blank spaces, is not the same as a VARCHAR data type column that is not padded with blank spaces. The following example illustrates this point:

  ```
  SQL> SHOW TABLE (COLUMNS) T1;
  Information for table T1

  Columns for table T1:
  Column Name                    Data Type        Domain
  -----------                    ---------        ------
  CHR                            CHAR(10)
  VARCHR                         VARCHAR(10)
  ```

```
SQL> INSERT INTO T1
cont> (CHR, VARCHR)
cont> VALUES ('abc', 'abc');
1 row inserted
SQL> --
SQL> SELECT CHR FROM T1 WHERE CHR LIKE 'abc';
0 rows selected
SQL> --
SQL> SELECT VARCHR FROM T1 WHERE VARCHR LIKE 'abc';
 VARCHR
 abc
1 row selected
```

In the previous example, the same string literal values are inserted into the CHR and VARCHR columns. However, the LIKE predicate returns different results because the CHAR data type pads the remainder of the string literal with seven blank spaces and the LIKE predicate does not. If you want to select the row in the CHR column, you need to issue the following SELECT command:

```
SQL> SELECT CHR FROM T1 WHERE CHR LIKE 'abc       ';  -- abc plus 7 spaces
 CHR
 abc
1 row selected
```

When you are declaring host variables for pattern matching, use the VARCHAR data type to avoid padding with blank spaces.

- When used with a column reference, the LIKE predicate expects a text data type for the pattern and does not convert a numeric data type to text.
- If automatic transaltion has not been enabled, the character set of the value expression, pattern, and escape character must be identical.
- If the character set of the value expression contains ASCII, you must use the ASCII percent sign (%) or underscore (_) as wildcard characters. For example, if the character set is DEC_KANJI, you must use the ASCII percent sign (%) or underscore (_) as wildcard characters. Table 2–34 shows the equivalent wildcard characters for each character set.
- If the character set of the value expression does not contain ASCII characters, you must use the percent sign or underscore characters from that character set to represent the wildcard characters.
- If you want the LIKE predicate to ignore the distinction between uppercase and lowercase characters, specify the IGNORE CASE keywords as part of a Boolean expression in the LIKE predicate. You cannot specify both

the IGNORE CASE and the ESCAPE keywords in the same Boolean expression.

For example, when you use the IGNORE CASE keywords to search for a character string that contains the * character, the * character works the same way as a wildcard % character. Therefore, you cannot use the LIKE predicate to search for the * character because you cannot use an ESCAPE keyword in a LIKE predicate when the IGNORE CASE keywords are used.

- SQL ignores the IGNORE clause if the character set of the value expression does not have uppercase and lowercase characters.

See Section 2.6 for details on value expressions. See Section 2.4 for details on literals and Section 2.2.13 for information on parameters.

Table 2–34 shows the wildcard characters for the supported character sets.

Table 2–34 Wildcard Characters

Character Set	Underscore	Percent
DEC_MCS	%X'5F'	%X'25'
BIG5	%X'A1C4'	%X'A248'
AL24UTFFSS	%X'5F'	%X'25'
ISOLATINARABIC	%X'5F'	%X'25'
ASCII	%X'5F'	%X'25'
DOS_LATIN1	%X'5F'	%X'25'
DOS_LATINUS	%X'5F'	%X'25'
GB18030	%X'5F'	%X'25'
ISOLATIN1	%X'5F'	%X'25'
ISOLATIN9	%X'5F'	%X'25'
ISOLATINCYRILLIC	%X'5F'	%X'25'
ISOLATINGREEK	%X'5F'	%X'25'
ISOLATINHEBREW	%X'5F'	%X'25'
DEVANAGARI	%X'5F'	%X'25'
KATAKANA	%X'5F'	%X'25'
KANJI	%X'A1B2'	%X'A1F3'
DEC_KANJI	%X'5F'	%X'25'

(continued on next page)

Table 2–34 (Cont.) Wildcard Characters

Character Set	Underscore	Percent
HANZI	%X'A3DF'	%X'A3A5'
DEC_HANZI	%X'5F'	%X'25'
KOREAN	%X'A3DF'	%X'A3A5'
DEC_KOREAN	%X'5F'	%X'25'
HANYU	%X'A2A8'	%X'A2A5'
DEC_SICGCC	%X'5F'	%X'25'
DEC_HANYU	%X'5F'	%X'25'
SHIFT_JIS	%X'5F'	%X'25'
TACTIS	%X'5F'	%X'25'
HEX	%X'3546'	%X'3235'
UNICODE	%X'005F'	%X'0025'
UTF8	%X'5F'	%X'25'
WIN_ARABIC	%X'5F'	%X'25'
WIN_GREEK	%X'5F'	%X'25'
WIN_CYRILLIC	%X'5F'	%X'25'
WIN_HEBREW	%X'5F'	%X'25'
WIN_LATIN1	%X'5F'	%X'25'

Example 1: Using the LIKE predicate and arguments without spaces

```
SQL> -- Notice that the LAST_NAME column
SQL> -- in the EMPLOYEES table has 14 characters:
SQL> SHOW TABLE EMPLOYEES
Information for table EMPLOYEES

Comment on table EMPLOYEES:
personal information about each employee

Columns for table EMPLOYEES:
Column Name                     Data Type        Domain
-----------                     ---------        ------
EMPLOYEE_ID                     CHAR(5)          ID_DOM
LAST_NAME                       CHAR(14)         LAST_NAME_DOM
         .
         .
         .
```

```
SQL> -- That means the following statement will not find the row for
SQL> -- Toliver because the LIKE predicate does not pad arguments with
SQL> -- blanks, and the character string "Toliver" only has 7 characters.
SQL> --
SQL> SELECT LAST_NAME FROM EMPLOYEES
cont>    WHERE LAST_NAME LIKE 'Toliver';
0 rows selected
SQL> --
SQL> -- To find the row for Toliver using the LIKE predicate, use the
SQL> -- percent sign wildcard. Note that you can also explicitly pad
SQL> -- the string by typing 7 underscore characters following the
SQL> -- word Toliver.
SQL> --
SQL> SELECT LAST_NAME FROM EMPLOYEES
cont>   WHERE LAST_NAME LIKE 'Toliver%';
 LAST_NAME
 Toliver
1 row selected
SQL> --
```

Example 2: Using the LIKE predicate and the percent sign wildcard character

If one percent sign wildcard is used in conjunction with an underscore character, the query retrieves all last names where *on* appears immediately after the first character in the name. In this example, the underscore represents the first character of the names, and a percent sign wildcard represents any characters following *on*:

```
SQL> SELECT  DISTINCT LAST_NAME
cont>   FROM EMPLOYEES
cont>   WHERE LAST_NAME LIKE '_on%';
 LAST_NAME
 Connolly
 Lonergan
2 rows selected
```

If two percent sign wildcards are used, this query retrieves all last names where *on* appears in any position in the name. The percent sign wildcards represent any characters preceding and following *on*:

```
SQL> SELECT DISTINCT LAST_NAME
cont>   FROM EMPLOYEES
cont>   WHERE LAST_NAME LIKE '%on%';
 LAST_NAME
 Burton
 Canonica
 Clairmont
 Clinton
 Connolly
 Goldstone
    .
    .
    .
```

Example 3: Using the LIKE predicate with numeric data types

The LIKE predicate also works with numeric data types, but compares the values with string literals. Find the salaries that begin with the number 3.

```
SQL> SELECT SALARY_AMOUNT
cont>   FROM SALARY_HISTORY
cont>   WHERE SALARY_AMOUNT LIKE '3%';
%SQL-I-NUMCMPTXT, Numeric column will be compared with string literal as text
 SALARY_AMOUNT
      30598.00
      30880.00
         .
         .
         .
```

This example is not another way of finding all the salaries in the range between $30,000 and $39,999. If the SALARY_AMOUNT column contained the value 398, the query would have retrieved it as well.

Example 4: Matching patterns with the LIKE predicate

Find the names of employees in which the letters *on* appear last in the last name. Because the column LAST_NAME is 14 characters long (CHAR(14)) and the matching pattern in this example specifies 7 explicit spaces after the sequence *on*, this query retrieves only 7-character names that end with *on*.

```
SQL> SELECT DISTINCT LAST_NAME
cont>   FROM EMPLOYEES
cont>   WHERE LAST_NAME LIKE '%on       ';
 LAST_NAME
 Clinton
 Jackson
 Johnson
3 rows selected
```

Increasing the number of explicit blank spaces in the matching pattern causes the query to retrieve shorter last names ending with the letters *on*. Decreasing the number of explicit blank spaces in the matching pattern causes the query to retrieve longer last names ending with the letters *on*.

Example 5: Using escape characters with the LIKE predicate

Find all the employees with a salary increase in the REMARKS column of their SALARY_HISTORY record.

```
SQL> SELECT LAST_NAME, REMARKS
cont>   FROM SALARY_HISTORY
cont>   WHERE REMARKS
cont>   LIKE '%&% increase%'      ESCAPE '&';
 LAST_NAME      REMARKS
 Clinton        10% increase
 Jackson        10% increase
 Johnson        10% increase
3 rows selected
```

The LIKE predicate needs to search for a character string containing the percent sign (%), which is a wildcard character. To search for the percent sign itself, the query designates the ampersand (&) as an escape character that stands for the percent sign in the search string.

Example 6: Matching case-sensitive patterns with the LIKE predicate

Find the last names of employees in which the characters *boy* are found. Because the LIKE predicate is case sensitive and the LAST_NAME columns were entered in uppercase and lowercase characters, it finds rows matching *%Boy%*. However, it will not find any rows matching *%BOY%*.

```
SQL> SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME
cont>   FROM EMPLOYEES E
cont>   WHERE LAST_NAME LIKE '%Boy%';
 EMPLOYEE_ID   LAST_NAME        FIRST_NAME
 00244         Boyd             Ann
 00226         Boyd             James
2 rows selected
SQL> --
SQL> SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME
cont> FROM EMPLOYEES E
cont> WHERE LAST_NAME LIKE '%BOY%';
0 rows selected
```

```
SQL> --You can also use the IGNORE CASE clause to get a case-insensitive match.
SQL> --Note that the % wildcard is used to search for padded
SQL> --characters that might be stored in the LAST_NAME column.
SQL> --
SQL> SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME
cont>   FROM EMPLOYEES
cont>   WHERE LAST_NAME LIKE '%BOY%' IGNORE CASE;
 EMPLOYEE_ID   LAST_NAME        FIRST_NAME
 00244         Boyd             Ann
 00226         Boyd             James
2 rows selected
```

Example 7: Using the LIKE predicate with a column reference

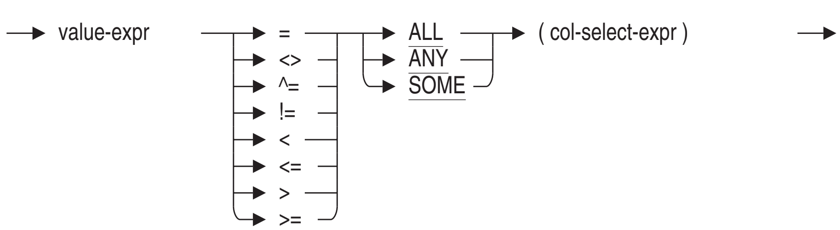
The following example demonstrates how to use the LIKE predicate to match a column reference:

```
SQL> CREATE TABLE PATTERN
cont> (A  CHAR VARYING(10));
SQL> --
SQL> INSERT INTO PATTERN
cont> (A)
cont> VALUES
cont> ('T%');
SQL> --
SQL> SELECT LAST_NAME FROM EMPLOYEES, PATTERN
cont>  WHERE LAST_NAME LIKE
cont>  (SELECT A FROM PATTERN);
 EMPLOYEES.LAST_NAME
 Tarbassian
 Toliver
2 rows selected
SQL>
```

2.7.9 Quantified Predicate

A quantified predicate compares a value with a collection of values. It has the same form as a basic predicate except that the second operand must be a column select expression preceded by an ALL, ANY, or SOME comparison operator.

quantified-predicate =



See Section 2.6 for details on value expressions. See Section 2.8.2 for details on column select expressions.

Table 2–35 describes the value of the result based on the comparison of values for the quantified predicate.

Table 2–35 Quantified Predicate Result Table

Comparison of Values	Result
ALL Quantifier	
If all comparisons are True	True
If any comparisons are False	False
If no comparisons are made	True
Otherwise	Unknown
SOME and ANY Quantifiers	
If any comparisons are True	True
If all comparisons are False	False
If no comparisons are made	True
Otherwise	Unknown

Examples: Using the quantified predicate

Example 1: Using the quantified predicate with the ALL operator

The following example uses the ALL operator in a quantified predicate to find the oldest and youngest employees:

```
SQL> SELECT FIRST_NAME, LAST_NAME, BIRTHDAY FROM EMPLOYEES
cont>         WHERE
cont>         BIRTHDAY <= ALL (SELECT BIRTHDAY FROM EMPLOYEES)
cont>         OR
cont>         BIRTHDAY >= ALL (SELECT BIRTHDAY FROM EMPLOYEES);
 FIRST_NAME   LAST_NAME        BIRTHDAY
 Rick         O'Sullivan       12-JAN-1923
 James        Stornelli        10-JAN-1960
2 rows selected
```

Example 2: Comparing quantified predicates

The following example uses the ANY operator in a quantified predicate to find the names of employees with college degrees. This query could be worded as "Select all the names and identification numbers of employees where there is at least one corresponding row in the DEGREES table."

```
SQL> SELECT  E.LAST_NAME, E.EMPLOYEE_ID
cont> FROM    EMPLOYEES E
cont>         WHERE E.EMPLOYEE_ID
cont>                 = ANY            -- same as = SOME
cont>                 (SELECT D.EMPLOYEE_ID
cont>                         FROM DEGREES D);
```

From the previous example, you might expect that a similar query using the <> ANY quantified predicate would return the names of employees who did not have college degrees. The following example shows that such a query retrieves all the rows in the EMPLOYEES table instead:

```
SQL>
SQL> -- Notice that the <> ANY predicate does *not* find
SQL> -- the names of employees without degrees. Instead, it retrieves
SQL> -- the names of employees from the EMPLOYEES table whose employee
SQL> -- IDs are not equal to any *one* of the EMPLOYEE_ID values in
SQL> -- the DEGREES table (in other words, every row in the EMPLOYEES table).
SQL> --
SQL> SELECT E.LAST_NAME, E.EMPLOYEE_ID
cont> FROM EMPLOYEES E
cont> WHERE E.EMPLOYEE_ID <> ANY
cont>   (SELECT D.EMPLOYEE_ID
cont>           FROM DEGREES D);
 LAST_NAME        EMPLOYEE_ID
 Toliver          00164
 Smith            00165
   .                .
   .                .
   .                .
SQL>
SQL> -- To retrieve the names of employees without degrees, use the
SQL> -- NOT Boolean operator to negate the = ANY quantified predicate.
SQL> -- (Another way to retrieve this information is with the
SQL> -- NOT EXISTS predicate.)
SQL> SELECT LAST_NAME, EMPLOYEE_ID
cont>  FROM EMPLOYEES
cont>  WHERE NOT (EMPLOYEE_ID = ANY
cont>   (SELECT EMPLOYEE_ID FROM DEGREES));
 LAST_NAME        EMPLOYEE_ID
 Goldstone        00178
1 row selected
```

2.7.10 SINGLE Predicate

The SINGLE predicate tests whether or not the result table specified in the column select expression has exactly one row. If it has exactly one row, SQL evaluates the SINGLE predicate as true. If the result table has zero rows or more than one row, the predicate is false.

SQL evaluates the NOT SINGLE predicate as true if the result table specified in the select expression has zero rows or more than one row.

The SINGLE and NOT SINGLE predicates cannot be unknown.

The SINGLE predicate has the following form:

single-predicate =

```
──► SINGLE ──► ( select-expr )        ────►
```

Because it checks only for the existence of rows, a SINGLE predicate does not require that the result table from its column select expression be a single column wide (see Section 2.8.2 for details on column select expressions). For SINGLE predicates, an asterisk (*) wildcard in the column select expression can refer to a multicolumn table (as in the following example).

Example: Using the SINGLE predicate

The following example determines which employees have one degree:

```
SQL> SELECT E.LAST_NAME, E.EMPLOYEE_ID
cont> FROM EMPLOYEES E
cont> WHERE SINGLE
cont> --
cont> -- Notice that the column select expression uses a wildcard,
cont> -- which is valid for multicolumn tables in SINGLE predicates:
cont> --
cont> (SELECT * FROM DEGREES D
cont>    WHERE D.EMPLOYEE_ID =
cont>          E.EMPLOYEE_ID);
 LAST_NAME        EMPLOYEE_ID
 Smith            00165
 Wood             00170
 Peters           00172
.
.
.
```

2.7.11 STARTING WITH Predicate

The STARTING WITH predicate tests whether or not the first characters of the first value expression match those specified in the second value expression. The STARTING WITH predicate has the following form:

starting-with-predicate =

Because the STARTING WITH predicate is case sensitive, it searches for uppercase characters and does not include lowercase characters for the DEC Multinational Character Set; the reverse is also true. For example, `STARTING WITH 'Ç'` retrieves a set of records different from those retrieved by `STARTING WITH 'ç'`.

The STARTING WITH predicate is sensitive to diacritical markings used in the DEC Multinational Character Set. Therfore, *a* matches *A*, but neither matches *á*, *à*, *ä*, *Á*, *À*, *Â* and so on.

In Spanish, *ch* and *ll* are treated as if they were unique single characters. For example, if a domain is defined with the collating sequence SPANISH, then `STARTING WITH 'c'` does not retrieve the word *char*, but retrieves the word *cat*.

Example: Using the STARTING WITH predicate

The following example shows how the STARTING WITH predicate displays last names and postal codes of employees whose postal codes begin with 030:

```
SQL> SELECT E.LAST_NAME, E.POSTAL_CODE FROM EMPLOYEES E
cont> WHERE E.POSTAL_CODE STARTING WITH '030';
 LAST_NAME        POSTAL_CODE
 Nash             03044
   .
   .
   .
Johnson          03055
 Klein            03055
9 rows selected
```

2.7.12 UNIQUE Predicate

The UNIQUE predicate is used to determine if duplicate rows exist in the result table of a column select expression. Note that the UNIQUE predicate (in compliance with the SQL language standard) ignores rows with a NULL column value and ensures uniqueness for the other column values. Contrast this with the SINGLE predicate, which considers a single column value of NULL as a match for any other NULL value in the same column.

unique-predicate =

```
──► UNIQUE ──► ( col-select-expr )        ──►
```

If any two rows in the expression are equal to one another, the UNIQUE predicate evaluates to false.

The following example determines those cities in which one and only one employee from the EMPLOYEES database lives.

```
SQL> SELECT E.LAST_NAME, E.CITY FROM EMPLOYEES E
cont> WHERE UNIQUE
cont> (SELECT * FROM EMPLOYEES EMP
cont> WHERE EMP.CITY=E.CITY);
 LAST_NAME        CITY
 Harrison         Boston
 Smith            Bristol
 McElroy          Cambridge
 Kilpatrick       Marlow
 Sciacca          Munsonville
 Vormelker        Rochester
 Dement           Sanbornton
 Babbin           Sanbornville
 Keisling         Twin Mountain
 Ziemke           Winnisquam
 Johnston         Wolfeboro
11 rows selected
```

2.8 Select Expressions and Column Select Expressions

Two fundamental elements of SQL syntax are the select expression and the column select expression. Select expressions specify result tables and column select expressions return a scalar value. A **result table** is an intermediate table derived from some combination of the table references identified in the FROM clause of the expression. A table reference is a base table, view, derived table, or a joined table.

Select expressions are the basis for the SELECT, DECLARE CURSOR, FOR cursor loop, CREATE VIEW, and INSERT . . . SELECT statements. Select expressions specify a result table to be retrieved from the database or to be stored in the database, and are derived from some combination of the table references identified in the FROM clause of the expression.

Column select expressions are select expressions that specify a one-column result table and can be nested within predicates and (if they specify a single value) value expressions.

Table 2–36 summarizes how select expressions and column select expressions are used with other statements. The remainder of this section describes select expressions and column select expressions in detail.

Table 2–36 Summary of Different Forms of the Select Statement

Form	Usage	Description	Also Called
SELECT statement	Least restrictive form, for general interactive or dynamic use. See the SELECT Statement: General Form.	Select expression.	
Select expression	Basic form of SELECT. Used in: • SELECT • DECLARE CURSOR • CREATE VIEW • INSERT . . . SELECT	See Section 2.8.1	Query specification (ANSI/ISO SQL)

(continued on next page)

Table 2–36 (Cont.) Summary of Different Forms of the Select Statement

Form	Usage	Description	Also Called
Column select expression	SELECT expression within predicates and used as value expression.	Select expression without select list. Within a predicate, result table must be no more than one column wide (except for EXISTS and SINGLE predicates). As a value expression, result table must contain a single value.	Subquery (ANSI/ISO SQL)
Singleton select	SELECT statement within host language programs to assign single row of values to host language variables. See SELECT Statement: Singleton Select.	Select expression with INTO clause after SELECT list. Result table must be no more than one row long.	Select statement (ANSI/ISO SQL)

2.8.1 Select Expressions

Using select expressions, you can define four types of result tables:

- Simple tables
- Aggregate tables
- Joined tables
- Derived tables

A **simple table** result is generated from a single table or view and usually includes the special DBKEY column. The rows of such tables can be updated, if privileges allow.

An **aggregate table** is a virtual table formed by the GROUP BY clause (allowing multi-row results) or an aggregate (or statistical) expression without allowing a GROUP BY (a single row result).

A table reference is a base table, view, derived table, or a joined table.

A **derived table** is a named virtual table that represents data obtained through the evaluation of a select expression. A derived table is named by the specified correlation name. References to a derived table and its columns can be made within the query using the correlation name. A derived table is similar to a view in that a view is also a virtual table represented by the select expression used to define it. Therefore, a derived table is like a view whose definition is specified within the FROM clause.

A **joined table** is a virtual table that represents data obtained through the joining of two table references. The type of join between the two table references can be either CROSS, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, or FULL OUTER JOIN. You need to use the joined table syntax to specify an outer join operation.

There are two types of joined tables:

- Qualified join
- Cross join

See the following syntax and arguments for more information on joined and derived tables.

See Appendix G for information on Oracle style join tables.

Environment

You can use select expressions, by themselves or as part of other SQL statements, in interactive SQL or in host language programs.

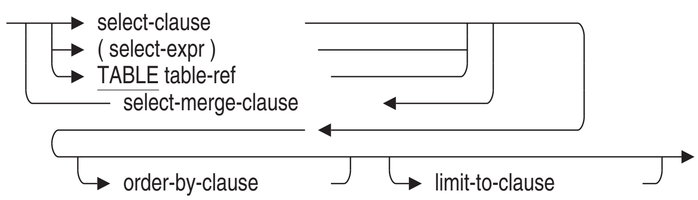
SQL evaluates the arguments in a select clause in the following order:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. Select list
6. ORDER BY
7. LIMIT TO

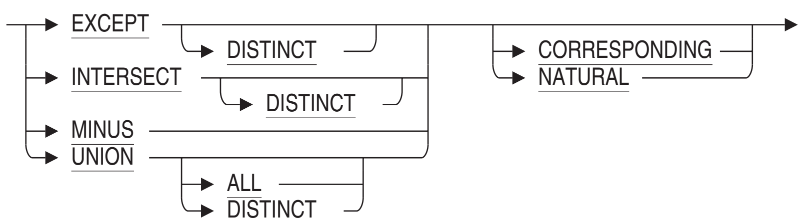
After each of these clauses, SQL produces an intermediate result table that is used in evaluating the next clause. The optimizer finds the fastest way of doing this without changing the result.

Format

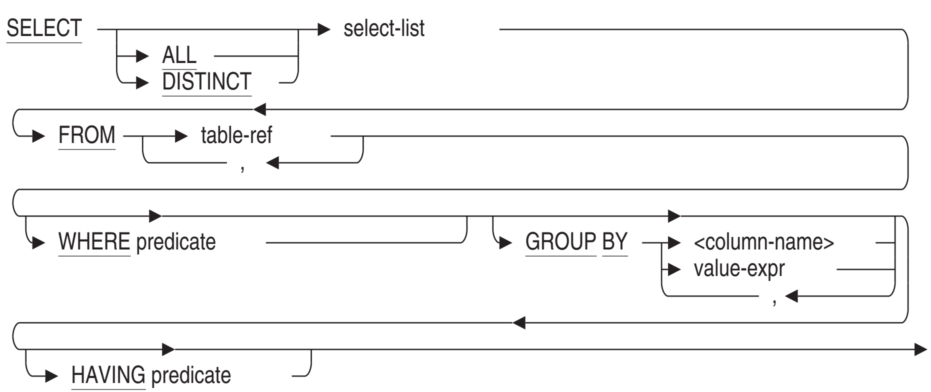
select-expr =



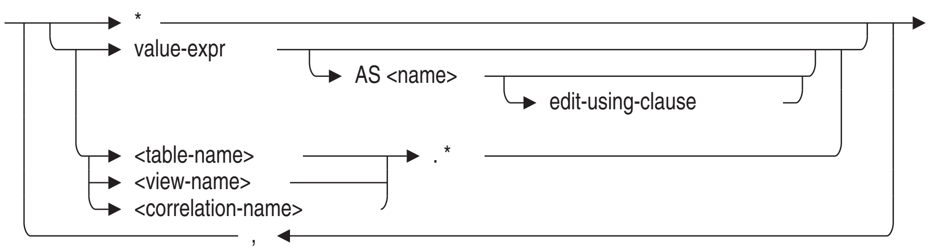
select-merge-clause =



select-clause =



select-list =



edit-using-clause =

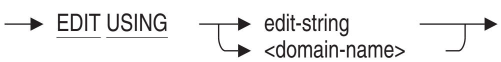
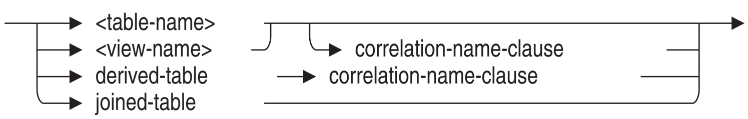
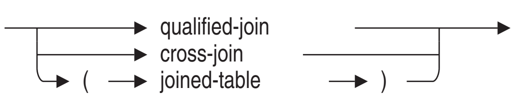


table-ref =

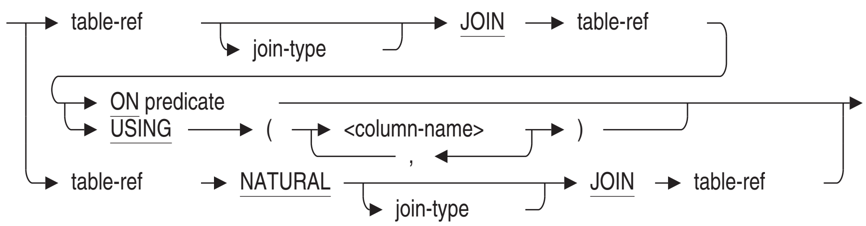


derived-table =

joined-table =

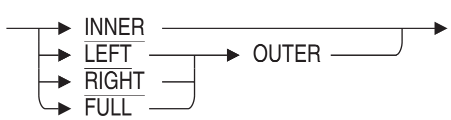


qualified-join =

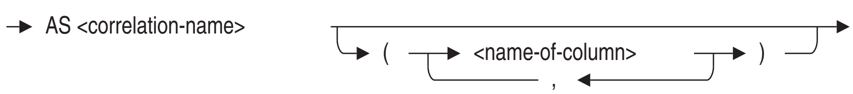


cross-join =

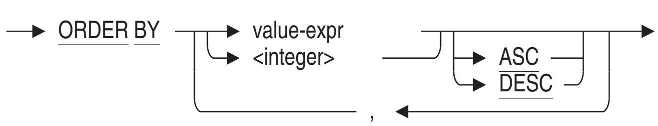
join-type =



correlation-name-clause =

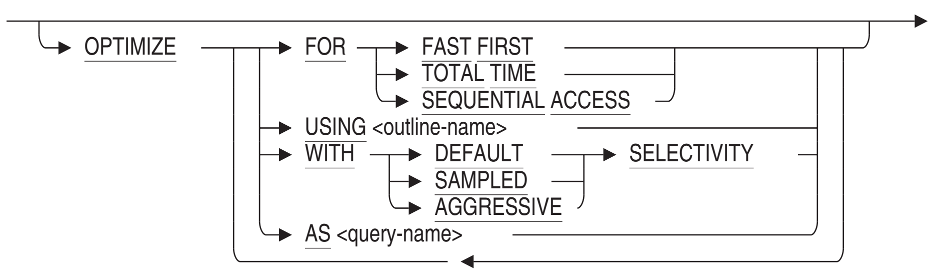


order-by-clause =



limit-to-clause =

optimize-clause =



Arguments

AS name

You can, optionally, give a column a name that might not otherwise be named using the AS clause. For example:

```
SQL> SELECT JOB_CODE AS JOB,
cont> MAXIMUM_SALARY - MINIMUM_SALARY AS RANGE
cont> FROM JOBS
cont> WHERE JOB_CODE LIKE 'S%';
 JOB                    RANGE
 SANL                20000.00
 SCTR                15000.00
 SPGM                25000.00
3 rows selected
```

You can use asterisks (*) as wildcards in a select list.

To use delimited identifiers, you must specify the SQL99 or similar dialect, or use the SET QUOTING RULES statement for these dialects.

ASC
DESC

Determines whether the values for sort keys are sorted in ascending or descending order.

If you do not specify the sort order for the sort key, the default order is ascending.

If your dialect is set to 'SQLV40' (the default dialect) then the sort order is inherited from the preceding sort key.

correlation-name-clause

You can specify a correlation name following a table or a view, and you must specify a correlation name for a derived table in the FROM clause to qualify column names in other parts of the select expression. If you do not explicitly specify a correlation name, SQL implicitly specifies the table name or view name as a correlation name. The same correlation name may not be specified more than once, either explicitly or implicitly.

The correlation name may also rename columns when specified with a derived table. Therefore, the number of columns in the table to the left of the correlation name must match the number of columns specified to the right of the correlation name.

CORRESPONDING

The UNION, EXCEPT, MINUS, and INTERSECT operators can be followed by the keyword CORRESPONDING. This causes the two select lists of the select-merge-clause to be compared by name. Only those column names which appear in both lists are retained for the resulting query table.

The name is either the column name, or the name provided by the AS clause. If there are no names in common, or a column name appears more than once in a select list then an error is reported.

CROSS JOIN

Combines all rows of the left-specified table reference to all rows of the right-specified table reference in the result. A cross join is a Cartesian product between two table references. A cross join is similar to the basic join expression but without the WHERE clause. This is also called a **Cartesian product**. Following is an example of the basic join expression using the comma-separated syntax:

```
SQL> SELECT *
cont>   FROM TABLE1, TABLE2;
```

Using the CROSS JOIN clause, the previous example would appear as follows:

```
SQL> SELECT *
cont>   FROM TABLE1 CROSS JOIN TABLE2;
```

EDIT USING edit-string
EDIT USING domain-name

Associates an edit string with a value expression. When a domain-name is specified, the edit string defined for that domain is used. This clause overrides any EDIT STRING defined for the columns or variables in the query. This clause is valid in interactive SQL only.

The following example illustrates the use of this clause.

```
SQL> set quoting rules 'SQL99';
SQL>
SQL> create domain MONEY integer(2) edit string '$$$,$$$,$$9';
SQL>
SQL> select
cont>     last_name as "Last Name",
cont>     employee_id,
cont>     birthday as "Birthday" edit using 'YYYYBDDBMMM',
cont>     (select salary_amount
cont>      from salary_history sh
cont>      where sh.employee_id = e.employee_id
cont>        and salary_end is null) as "Salary" edit using MONEY
cont> from employees e
cont> where e.employee_id < '00167';
 Last Name        EMPLOYEE_ID   Birthday           Salary
 Toliver          00164         1947 28 Mar       $51,712
 Smith            00165         1954 15 May       $11,676
 MacMullan        00166         1954 20 Mar       $18,497
3 rows selected
```

EXCEPT
EXCEPT DISTINCT

The EXCEPT DISTINCT operator is used to create a result table from the first select expression except for those row values that also occur in the second select expression.

DISTINCT is the default so EXCEPT and EXCEPT DISTINCT are identical operations. EXCEPT conforms to the ANSI and ISO SQL:1999 Database Language Standard.

Note

EXCEPT is not commutative. That is, A EXCEPT B may result in a different set of rows from B EXCEPT A.

FROM derived-table

A derived table is a named virtual table containing data obtained through the evaluation of the select expression in the FROM clause. The derived table is named by specifying the correlation name.

You must specify a correlation name for a derived table. This may determine which column names the user can specify in the select list or subsequent clauses. The select list and subsequent clauses can reference only the correlation name and the column names of the derived table and cannot reference the table or column names that defined the derived table.

Following is an example of a derived table using the personnel database. This example finds all departments that have less than 3 rows in the JOB_HISTORY table.

```
SQL> SELECT *
cont> FROM (SELECT DEPARTMENT_CODE, COUNT(*)
cont>       FROM JOB_HISTORY
cont>       WHERE JOB_END IS NULL
cont>       GROUP BY DEPARTMENT_CODE)
cont>       AS DEPT_INFO (D_CODE, D_COUNT)
cont> WHERE D_COUNT < 3;
 D_CODE       D_COUNT
 ENG                2
 MCBS               1
 MSMG               1
 MTEL               2
 PERS               2
 SUSA               2
6 rows selected
```

FROM joined-table
A joined table represents a join between two table references specified in the FROM clause.

There are two types of joined tables:

- Qualified join—syntax contains either an implicit or explicit predicate
- Cross join—syntax does not contain a predicate

A table can be joined to itself or joined to other tables. When an outer join is specified in the joined-table expression, you can use the parentheses to explicitly define the join order. If only inner or cross joins are specified in the joined-table expression, the use of parentheses does not affect the join order. SQL tries all possible join orders to find the most efficient order for the query. If outer joins are specified in the joined-table expression, the join order is determined first by any existing parentheses and then by the left-to-right rule.

The table or correlation names specified in the joined-table expression can be referenced by the outer select expression.

FROM table-name
FROM view-name
Identifies the tables and views that SQL uses to generate the result table. If you name more than one table or view, SQL joins them to create an intermediate result table.

FULL OUTER JOIN
Preserves all rows from the left-specified table reference and all rows from the right-specified table reference in the result. NULL appears in any column that does not have a matching value in the corresponding column. For example:

```
SQL> SELECT *
cont> FROM TABLE1 FULL OUTER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1;
   TABLE1.C1     TABLE1.C2     TABLE2.C1   TABLE2.C4
          10            15            10   AA
        NULL          NULL            15   BB
          20            25            20   CC
          30            35          NULL   NULL
4 rows selected
```

You must specify at least one equijoin condition in the ON clause of a FULL OUTER JOIN clause. This restriction does not apply to a FULL OUTER JOIN clause with the USING clause or to the NATURAL FULL OUTER JOIN clause.

An **equijoin** matches values in columns from one table with the corresponding values of columns in another table implicitly using an equal (=) sign.

GROUP BY value-expr
Indicates the value expressions that SQL uses for organizing the intermediate result table from the WHERE clause, if specified, or the FROM clause. These groups of rows containing the same value are also called **control breaks**.

For the first expression specified in the GROUP BY clause, SQL orders the rows of the preceding intermediate result table into groups whose rows all have the same value for the specified expression. If a second expression is specified in the GROUP BY clause, SQL then groups rows within each main group by values of the second expression. SQL groups any additional columns in the GROUP BY clause in a similar manner.

All null values for a column name in the GROUP BY clause are grouped together.

Each group is treated as the source for the values of a single row of the result table.

Because all rows of a group have the same value for the value expression specified in the GROUP BY clause, references within value expressions or predicates to that column specify a single value.

HAVING predicate
Specifies a predicate that SQL evaluates to generate an intermediate result table. SQL evaluates the predicate for each group of the intermediate result table created by a preceding clause. The groups of that table for which the

predicate is true become another intermediate result table to which SQL applies the select list for evaluation.

If the clause preceding the HAVING clause is a GROUP BY clause, then the predicate is evaluated for each group in the intermediate result table. The HAVING clause affects groups just as the WHERE clause affects individual rows.

If the HAVING clause is not preceded by a GROUP BY clause, SQL evaluates the predicate for all the rows in the intermediate result table as a single group.

SQL restricts which expressions you can specify in the predicate of a HAVING clause. A column name or expression in a HAVING predicate must meet one of the following criteria:

- It must also appear in the GROUP BY clause.
- It must be specified within an aggregate function.
- It must be an outer reference (possible only if the HAVING clause is part of a column select expression).

For instance, the following statement is invalid. It has a HAVING clause without a GROUP BY clause, which means that any column names in the HAVING clause must be part of a function (because there is no outer query, the column names cannot be outer references).

```
SQL> SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES
cont> HAVING FIRST_NAME = 'Bob';
%SQL-F-NOTGROFLD, Column FIRST_NAME cannot be referred to in
the select list or HAVING clause because it is not in the GROUP BY clause
```

INNER JOIN

Combines all rows of the left-specified table reference to matching rows in the right-specified table reference. For example:

```
SQL> SELECT *
cont> FROM TABLE1 INNER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1
cont> AND C2 BETWEEN 25 AND 35;
   TABLE1.C1     TABLE1.C2     TABLE2.C1   TABLE2.C4
          10            15            10   AA
          20            25            20   CC
2 rows selected
```

Both TABLE1 and TABLE2 are exposed in the remainder of the select clause and, therefore, can be used to qualify columns from either table reference.

```
SQL> SELECT *
cont> FROM TABLE1 INNER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1
cont> WHERE TABLE1.C1 = 10;
   TABLE1.C1     TABLE1.C2     TABLE2.C1     TABLE2.C4
          10            15            10            AA
1 row selected
```

If INNER JOIN is specified in the joined-table expression, it implies any join ordering of the table references. For example, A INNER JOIN B INNER JOIN C is equivalent to A INNER JOIN C INNER JOIN B. In general, any permutation of table references A, B, and C in an inner join table expression produces the same result. Further, SELECT * FROM A INNER JOIN B ON P1 INNER JOIN C ON P2 is equivalent to the syntax SELECT * FROM A, B, C WHERE P1 AND P2.

INTERSECT
INTERSECT DISTINCT
The INTERSECT DISTINCT operator is used to create a result table from the first select expression for those row values that also occur in the second select expression.

DISTINCT is the default so INTERSECT and INTERSECT DISTINCT are identical operations. INTERSECT conforms to the ANSI and ISO SQL:1999 Database Language Standard.

Note

In general INTERSECT is commutative. That is, A INTERSECT B results in the same set of rows from B INTERSECT A. This is demonstrated by the examples below. However, care should be taken when using LIMIT TO within the different branches of the INTERSECT as this will make the result non deterministic because of possible different solution strategies employed by the Rdb optimizer.

LEFT OUTER JOIN
Preserves all rows in the left-specified table reference and matches to rows in the right-specified table reference in the result. NULL appears in columns where there is no match in the right-specified table. For example:

```
SQL> SELECT *
cont> FROM TABLE1 LEFT OUTER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1;
   TABLE1.C1     TABLE1.C2     TABLE2.C1   TABLE2.C4
          10            15            10   AA
          20            25            20   CC
          30            35          NULL   NULL
3 rows selected
```

Basically, outer joins are an inner join with a union adding NULL to all unmatched rows. Notice that the LEFT OUTER JOIN example results are the same as the INNER JOIN example results plus the unmatched row.

The search condition specified in the ON clause is used to construct the outer join result. In addition to the join predicates, you can specify selection predicates and subqueries in the ON clause. For example:

```
SQL> SELECT *
cont> FROM TABLE1 LEFT OUTER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1
cont> AND C2 BETWEEN 25 AND 35;
   TABLE1.C1     TABLE1.C2     TABLE2.C1   TABLE2.C4
          10            15          NULL   NULL
          20            25            20   CC
          30            35          NULL   NULL
3 rows selected
```

A select condition in the ON clause reduces the inner join result. The left outer join result contains the inner join result plus each row from TABLE1 that did not match a row in TABLE2 and was extended with NULL.

In contrast, the result from the following example uses the same selection condition but with the WHERE clause:

```
SQL> SELECT *
cont> FROM TABLE1 LEFT OUTER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1
cont> WHERE C2 BETWEEN 25 AND 35;
   TABLE1.C1     TABLE1.C2     TABLE2.C1   TABLE2.C4
          20            25            20   CC
          30            35          NULL   NULL
2 rows selected
```

In the previous example, the left outer join result is first constructed using the search condition in the ON clause. Then the selection condition in the WHERE clause is applied to each row in the outer join result to form the final result.

LIMIT TO row-limit ROWS

Specifies the number of rows in the result table. The row limit is a value expression.

If you specify both a select expression that can be updated and a LIMIT TO clause, the result table can be updated.

When specifying a row limit, you cannot use a value expression that refers to a column that is a column of the select expression.

If the value of the row limit that you specify is less than or equal to zero, the result table contains zero rows.

MINUS

The MINUS operator is a synonym for the EXCEPT DISTINCT operator and is provided for language compatibility with the Oracle RDBMS SQL language.

NATURAL JOIN

Performs an equijoin operation on the matching named columns of the specified tables. An equijoin matches values in columns from one table with the corresponding values of columns in another table implicitly using an equal (=) sign.

A NATURAL JOIN implicitly performs the following functions:

- Coalesces the common columns to condense the columns into a single column and, therefore, you cannot qualify the common column
- Performs an equijoin using common columns between table references

You cannot specify an explicit join condition if the NATURAL keyword is specified in the query. Following is an example of a natural join. Note the common column C1 is only shown once. Other types of join conditions return the common column as often as it occurs in the table's references.

```
SQL> SELECT *
cont> FROM TABLE1 NATURAL LEFT OUTER JOIN TABLE2;
          C1     TABLE1.C2   TABLE2.C4
          10            15          AA
          20            25          CC
          30            35        NULL
3 rows selected
```

The complexity of what the NATURAL LEFT OUTER JOIN is implicitly executing in the previous example is shown in the following example:

```
SQL> SELECT
cont>   COALESCE (TABLE1.C1, TABLE2.C1) AS C1,
cont>   TABLE1.C2, TABLE2.C4
cont> FROM TABLE1 LEFT OUTER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1;
          C1     TABLE1.C2   TABLE2.C4
          10           15          AA
          20           25          CC
          30           35        NULL
3 rows selected
```

The NATURAL keyword can be specified for INNER, LEFT OUTER, RIGHT OUTER, and FULL OUTER joins.

A natural join between two table references that do not share matching named columns results in a Cartesian product.

ON predicate
Specifies a search condition on which the join is based. The predicate can have columns from the two operands mentioned, or have outer references if it is in a subquery.

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:

- 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.

- 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;
```

- 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.

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).

ORDER BY integer
ORDER BY value-expr

Specifies the order of rows for the result table. SQL sorts the rows from the intermediate result table by the values of expressions specified in the ORDER BY clause. The **intermediate result table** is the result table SQL produces when it evaluates the preceding clause in the select expression (HAVING, GROUP BY, WHERE, or FROM).

You can refer to columns in the ORDER BY clause in two ways:

- By a value expression
- By column number, where the integer you specify indicates the left-to-right position of the column in the result table

You must use an integer to identify a column in the ORDER BY clause if that column in the select list is derived from a function, an arithmetic expression, or the result of a UNION, MINUS, EXCEPT, or INTERSECT operator.

Whether you identify expressions in an ORDER BY clause using a name or using a number, the expressions are called **sort keys.**

When you use multiple sort keys, SQL treats the first expression as the major sort key and successive keys as minor sort keys. That is, it first sorts the rows into groups based on the first value expression. Then, it uses the second value expression to sort the rows within each group, and so on. Unless you specify a sort key for every column in the result table, rows with identical values for the last sort key specified will be in arbitrary order.

The following example illustrates 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
   .
   .
   .
```

qualified-join
Qualifies and alters the result returned from the joined tables. There are several types of qualified joins:

- INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- NATURAL JOIN

For an INNER and OUTER JOIN, the result table is the combination of all columns of the first table reference to all the columns in the second table reference. For a NATURAL JOIN, the result table condenses common columns (that is, columns with the same name) between the table references. See the following arguments for more information.

RIGHT OUTER JOIN
Preserves all rows of the right-specified table reference and matches to rows in the left-specified table reference in the result. NULL appears in columns where there is no match in the left-specified table reference. For example:

```
SQL> SELECT *
cont> FROM TABLE1 RIGHT OUTER JOIN TABLE2
cont> ON TABLE1.C1 = TABLE2.C1;
   TABLE1.C1     TABLE1.C2     TABLE2.C1   TABLE2.C4
          10            15            10   AA
        NULL          NULL            15   BB
          20            25            20   CC
3 rows selected
```

Notice that the FULL OUTER JOIN example result is the same as the INNER JOIN example result plus the unmatched rows from TABLE1 and unmatched rows from TABLE2.

SELECT *
Tells SQL to use all the column results from the intermediate result table (namely, all the columns in all the table references referred to in the FROM clause). If the select expression contains a GROUP BY clause, SQL interprets the wildcard (*) as specifying only the expressions in the GROUP BY clause.

SELECT ALL
Specifies that duplicate rows should not be eliminated from the result table. ALL is the default.

SELECT DISTINCT
Specifies that SQL should eliminate duplicate rows from the result table.

SELECT name.*
Tells SQL to use all the columns in the table references referred to by the table name, view name, or correlation name. The name must be specified in the FROM clause of the select expression. You cannot mix this form of wildcard notation with SELECT *.

The number of columns you specify in the select list, either by using wildcards or by explicitly listing value expressions, is the number of columns in the result table. In

select-list
Identifies a list of value expressions (to be derived from the table references named in the FROM clause) for the final result table.

UNION
UNION DISTINCT
Merges the results of a select expression with another select expression into one result table by appending the values of columns in one table with the values of columns in other tables.

The following example extracts the EMPLOYEE_ID of current employees with a salary greater than $50,000 and with a Ph.D. Duplicate rows are eliminated from the result table:

```
SQL> SELECT EMPLOYEE_ID
cont>   FROM CURRENT_SALARY
cont>   WHERE SALARY_AMOUNT > 50000
cont> UNION
cont>   SELECT EMPLOYEE_ID
cont>     FROM DEGREES
cont>     WHERE DEGREE = 'PhD';
 EMPLOYEE_ID
 00164
 00166
 00168
 00169
 00172
 00182
   .
   .
   .
 00418
 00435
 00471
38 rows selected
```

UNION ALL
Specifies that duplicate rows should not be eliminated from the result table. By default, the UNION operator removes duplicate rows.

The following example returns duplicate rows from the result table:

```
SQL> SELECT LAST_NAME, SEX FROM EMPLOYEES WHERE LAST_NAME = 'Nash'
cont> UNION ALL
cont> SELECT LAST_NAME, SEX FROM EMPLOYEES WHERE LAST_NAME = 'Lapointe';
 LAST_NAME        SEX
 Nash             M
 Nash             M
 Lapointe         F
 Lapointe         F
4 rows selected
```

USING
Specifies the columns on which the join is based. Column names must be defined in both table references specified in the qualified join. The USING clause implies an equijoin condition between columns of the same name and creates a common column in the result.

```
SQL> SELECT *
cont> FROM TABLE1 LEFT OUTER JOIN TABLE2
cont> USING (C1);
          C1     TABLE1.C2   TABLE2.C4
          10            15          AA
          20            25          CC
          30            35        NULL
3 rows selected
```

The common columns are coalesced into a single column in the result in the previous example. Therefore, such columns cannot be qualified. You can reference the coalesced column in a query. For example:

```
SQL> SELECT *
cont> FROM TABLE1 LEFT OUTER JOIN TABLE2
cont> USING (C1)
cont> WHERE C1 BETWEEN 20 AND 30;
          C1     TABLE1.C2   TABLE2.C4
          20            25          CC
          30            35        NULL
2 rows selected
```

WHERE predicate

Specifies a predicate that SQL evaluates to generate an intermediate result table. SQL evaluates the predicate for each row of the intermediate result table created by the FROM clause. The rows of that table for which the predicate is true become another intermediate result table for later clauses in a select expression.

Column names specified in the predicate of the WHERE clause must either:

- Identify columns of the intermediate result table created by the FROM clause.
- Be an outer reference (possible only if the WHERE clause is part of a column select expression). See Section 2.2.4.2 for more information on outer references.

In general, the predicate in a WHERE clause cannot refer to an aggregate function. For instance, the following statement is invalid:

```
SQL> SELECT * FROM EMPLOYEES WHERE MAX(LAST_NAME) > 'X';
%SQL-F-INVFUNREF, Invalid function reference
```

See the Usage Notes in this section for a limited exception to this restriction.

Usage Notes

- You cannot specify a correlation name in a table reference that is the same as any other correlation name already specified in the containing FROM clause or that is the same as the table identifier of any table name exposed in the containing FROM clause. This restriction complies with the ANSI/ISO SQL standard.

  This restriction causes the error message that appears in the following example:

  ```
  SQL> SELECT * FROM JOBS, CURRENT_JOB JOBS;
  %SQL-F-CONVARDEF, Column qualifier JOBS is already defined
  SQL> --
  SQL> SELECT * FROM JOBS J, CURRENT_JOB J;
  %SQL-F-CONVARDEF, Column qualifier J is already defined
  ```

- The ordering of INNER, LEFT OUTER, RIGHT OUTER, and FULL OUTER joins is determined by the ON predicate. If you put your syntax inside parentheses, remember to also place the corresponding ON predicate inside those parentheses.
- For select expressions embedded in programs and modules (both stored and nonstored), SQL expands wildcards in select lists when it precompiles the program, not when the program runs.
- When specifying a column name in a select expression, if the column name is the same as a parameter, you must use a correlation name with the column name to distinguish it from the parameter.
- In general, the predicate in a WHERE clause cannot refer to an aggregate function. The only exception to this restriction is when the function in a WHERE clause has an outer reference as its argument. The only cases where this is possible are when the WHERE clause is a predicate for a column select expression that is:
  - A value expression in a select list item
  - Part of a predicate to a HAVING clause

    ```
    SQL> -- Display departments that have total current
    SQL> -- salaries greater than their projected budget:
    SQL> SELECT DEPARTMENT
    cont>  FROM CURRENT_INFO
    cont>  GROUP BY DEPARTMENT
    cont>  HAVING DEPARTMENT IN
    cont>    (SELECT DEPARTMENT_NAME
    cont>     FROM DEPARTMENTS
    cont>     WHERE SUM (CURRENT_INFO.SALARY) > BUDGET_PROJECTED);
    0 rows selected
    ```

- If you do not use the GROUP BY clause, the select list must either:
  - Be a list of aggregate functions only
  - Not contain any functions

  For example, SQL cannot evaluate the following query because it mixes a function and a column name:

  ```
  SQL> SELECT     EMPLOYEE_ID, AVG(SALARY_AMOUNT)
  cont> FROM      SALARY_HISTORY;
  %SQL-F-INVSELLIS, Select list cannot mix columns and
  functions without GROUP BY
  ```

  When you do specify a list of functions (without a GROUP BY clause), the result table generated by a select expression has only one row.

- If you use the GROUP BY or HAVING clauses, column names in the select list must either be:
  - In the GROUP BY clause also
  - Specified within an aggregate function

  For example, the following statement is invalid:

  ```
  SQL> SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES
  cont> GROUP BY LAST_NAME ;
  ```

  However, you can use a wildcard in the select list because SQL interprets the wildcard as referring only to the column names specified in the GROUP BY clause.

  For instance, the following statement is valid (in this case, the wildcard specifies only the LAST_NAME column):

  ```
  SQL> SELECT * FROM EMPLOYEES
  cont> GROUP BY LAST_NAME ;
  ```

- The characteristics of the columns in the result table of a select expression depend on how the columns were specified in the select list.
  - Columns in the result table derived directly from column names in the select list inherit the name, data type, and other characteristics of the source column as specified in the CREATE TABLE statement.
  - Columns derived through other value expressions in the select list can be named using the AS clause. They have data types that are the same as the result of the value expression.

    - Columns derived from literals in the select list do not allow null values. Columns derived from the COUNT function also do not allow null values. Columns derived from parameters allow null values if the parameter has an indicator parameter.

- The following restrictions apply when you use a value expression in a GROUP BY clause:
    - You must have a syntactically similar value expression in the select list.
    - The asterisk (*) is not supported in value expressions with GROUP BY.
- The EXCEPT DISTINCT operator can be rewritten to use the NOT ANY predicate. In fact the Rdb server implements EXCEPT DISTINCT in this way. Consider this example:

```
SQL> select manager_id from departments
cont> except distinct
cont> select employee_id from employees;
```

  This query could be rewritten as:

```
SQL> select manager_id
cont> from departments d
cont> where not exists (select *
cont>                   from employees e
cont>                   where e.employee_id = d.manager_id
cont>                      or (e.employee_id is null
cont>                          and d.manager_id is null));
```

  As you can see even for this simple query the EXCEPT format is easier to read. As the number of columns selected increases so does the complexity of the NOT EXISTS subquery.

- The INTERSECT DISTINCT operator can be rewritten to use the EXISTS predicate. In fact the Rdb server implements INTERSECT DISTINCT in this way. Consider this example which displays all managers which are also employees:

```
SQL> select manager_id from departments
cont> intersect distinct
cont> select employee_id from employees;
```

This query could be rewritten as:

```
SQL> select manager_id
cont> from departments d
cont> where exists (select *
cont>               from employees e
cont>               where e.employee_id = d.manager_id
cont>                  or (e.employee_id is null
cont>                      and d.manager_id is null));
```

As you can see even for this simple query the INTERSECT format is easier to read. As the number of columns selected increases so does the complexity of the EXISTS subquery.

- For both EXCEPT and INTERSECT all duplicate rows are eliminated. For the purposes of these operators a row is considered a duplicate if each value in the first select list is equal to the matching column in the second select list, or if both these columns are NULL.

  The duplicate matching semantics can be clearly seen in the rewritten queries which use NOT EXISTS and EXISTS.

2.8.2 Column Select Expressions

A **column select expression** is a select expression that specifies a one-column result table in one row and can be nested within predicates and (if they specify a single value) value expressions. Column select expressions cannot specify a list of select items. You can only specify one value in a select list.

Column select expressions are also called scalar expressions.

SQL accepts column select expressions as arguments to IN and quantified predicates, and more generally as value expressions.

- As arguments to IN and quantified predicates, column select expressions specify a collection of values to which SQL compares a value expression. Therefore, column select expressions as arguments to those predicates can return one or more values.
- As a type of value expression, column select expressions specify a single value. Therefore, a column select expression used as a value expression should not return more than one value. If it does, SQL generates the following error:

```
%RDB-E-MULTIPLE_MATCH, record selection criteria should identify
                       only one record; more than one record found
```

If a column select expression used as a value expression returns zero rows, SQL evaluates the value expression as null. The data type of a column select expression used as a value expression is the same as the data type of the column select expression's select item.

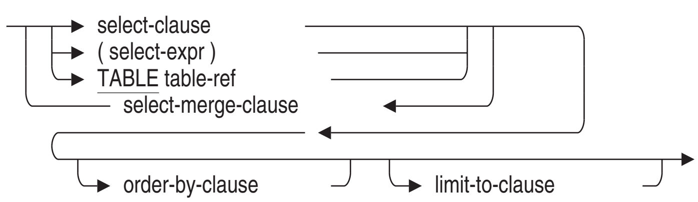
Environment

You can use column select expressions in interactive SQL or in host language programs.

Format

col-select-expr =

select-expr =



Arguments

select-expr

A column select expression is a select expression specifying only one value in the select list. See Section 2.8.1 for more information.

2.9 Context Structures

A **distributed transaction** groups more than one database or more than one database attachment together into one transaction, even if the databases are located on different nodes. The *Oracle Rdb7 Guide to Distributed Transactions* explains how to use Oracle Rdb with distributed transactions.

The **two-phase commit protocol** coordinates the activity of participants in a transaction to ensure that every required operation is completed before a transaction is made permanent, even if the transaction is a distributed transaction.

You can use the two-phase commit protocol when the DECdtm transaction manager software is installed and started on all nodes that are in the transaction. Use the two-phase commit protocol when your application starts a distributed transaction.

When you declare a context structure in an application, you must associate it with most executable SQL statements. This is true whether you use SQL module language or precompiled SQL, although the method you use to associate the context structure with SQL statements differs depending upon which compiler you choose.

However, you cannot associate a context structure with the following categories of executable statements:

- Statements that you cannot execute when a transaction is already started
- Statements that do not execute within the scope of a transaction and are independent of any transaction context
- Statements that you cannot use in transactions that were started by explicit calls to the transaction manager

You must use the USING CONTEXT clause to specify that an embedded SQL statement is part of a distributed transaction. For more information about using embedded SQL with distributed transactions, see Section 4.1. You must use the CONTEXT clause in a module language procedure to make SQL execute your procedure in the context of a distributed transaction. For more information about using SQL module language with distributed transactions, see Section 3.6.

The following restrictions apply when passing context structures:

- You cannot pass a context structure to the following SQL statements because you cannot execute them when a transaction is already started:
  - ALTER DATABASE
  - CREATE DATABASE
  - DROP PATHNAME
  - DROP DATABASE
- You cannot pass a context structure to the following SQL statements because they do not execute within the scope of a transaction, and they are independent of any transaction context:
  - CONNECT
  - DESCRIBE

- DISCONNECT
- Extended Dynamic DECLARE CURSOR
- RELEASE
- SET CATALOG
- SET CHARACTER LENGTH
- SET CONNECT
- SET DEFAULT CHARACTER SET
- SET DEFAULT DATE FORMAT
- SET DIALECT
- SET IDENTIFIER CHARACTER SET
- SET KEYWORD RULES
- SET LITERAL CHARACTER SET
- SET NAMES
- SET NATIONAL CHARACTER SET
- SET OPTIMIZATION LEVEL
- SET QUOTING RULES
- SET SCHEMA
- SET VIEW UPDATE RULES

* You cannot pass a context structure to the following SQL statements because they have been started by explicit calls to the transaction manager:
  - COMMIT
  - ROLLBACK

  (The DISCONNECT statement can be considered in this category as well as the previous category.)

Remember that you cannot associate a context structure with nonexecutable SQL statements. Moreover, you cannot pass a context structure to a multistatement procedure if that procedure contains a SET TRANSACTION, COMMIT, or ROLLBACK statement.

2.10 Database Options

By default, the SQL module language processor, or the SQL precompiler determines the type of database it attaches to from the type of database specified in compiling the program. If no database is used to compile the program, the program is processed for a database created with the most recent version of Oracle Rdb.

Specifying the database options in the DECLARE ALIAS statement overrides the default established in the precompiler or module processor command lines.

Table 2–37 shows the database options for interactive SQL, SQL module language processor, and SQL precompiler for OpenVMS.

Table 2–37 Database Options

SQL Module and Precompiler	Explanation
RDBVMS	Accesses a database created with the most recent version of Oracle Rdb.
RDB030	Accesses Oracle Rdb Version 3.0 databases.
RDB031	Accesses Oracle Rdb Version 3.1 databases.
RDB040	Accesses Oracle Rdb Version 4.0 databases.
RDB041	Accesses Oracle Rdb Version 4.1 databases.
RDB042	Accesses Oracle Rdb Version 4.2 databases.
RDB050	Accesses Oracle Rdb Version 5.0 databases.
RDB051	Accesses Oracle Rdb Version 5.1 databases.

In most cases, it is not necessary to specify a database option. For example, when you attach to an Oracle Rdb Version 7.0 database, SQL provides the V7.0 features.

However, you need to specify a database option when the database you attach to during compilation or precompilation has different features than the database against which the program is to run. You must specify a database option that provides the "lowest common denominator" of features for all the databases that the program intends to use at run time.

If no database is used during compilation of the program, the program is processed for a database created with the most recent version of Oracle Rdb. Therefore, if the resulting program is intended to run against a database other than the most recent version of Oracle Rdb, you must specify that version of the database option during compilation.

You can use any of the qualifiers listed in Table 2–37 to override the default database option.

2.11 Using Context Files with SQL Module Language and SQL Precompiler

You can use SQL context files with SQL module language just as you can use them with precompiled SQL. A **context file** is an SQL command procedure containing DECLARE statements that you want to apply when your program compiles and executes. Context files help improve the portability of compiled source files.

The format of a context file used with SQL module language is the same as the one used for precompiled SQL, with one exception. It is not necessary to end the DECLARE statements with a semicolon (;) when you use a context file with SQL module language. However, if you include the semicolon, you can use the context file with both module and precompiled SQL. When you use a context file, enter it as the second parameter on the command line.

Suppose an application contains a module that must be compiled using different SQL dialects. Rather than having two copies of the module and the problem of maintaining them in parallel, you can have one module and two context files. The module contains all the code, and each context file contains the dialect declaration statement. For example, assume that you need to compile the module TEST using two different dialects: SQL92 and MIA. You might create two context files:

- The context file TEST-SQL92 contains the following DECLARE MODULE statements:

  ```
  DECLARE MODULE
  DIALECT SQL92
  ```

- The context file TEST-MIA contains the following DECLARE MODULE statements:

  ```
  DECLARE MODULE
  DIALECT MIA
  ```

You can control the dialect you want to use by compiling the module with the appropriate context file:

- For TEST to use the SQL92 semantics, compile TEST using the TEST-SQL92 context file. The following example shows how to compile the module on OpenVMS:

```
$ SQL$MOD
SQL$MOD> TEST TEST-SQL92
```

- For TEST to use the MIA semantics, compile the module TEST using the TEST-MIA context file. The following example shows how to compile the module on OpenVMS:

```
$ SQL$MOD
SQL$MOD> TEST TEST-MIA
```

3

SQL Module Language

This chapter describes the SQL module language syntax, how to declare the length of character string parameters, equivalent SQL and host language data types, how to use context files with the SQL module language, and how to invoke the SQL module language processor and nonstored modules. It begins with a brief overview of the SQL module language and SQL module language processor.

For information about stored modules, see the CREATE MODULE Statement or the *Oracle Rdb Guide to SQL Programming*.

3.1 Overview of the SQL Module Language and Processor

The SQL module language and SQL module processor allow procedures that contain SQL statements to be called from any host language, including those not supported by the SQL precompiler.

The SQL module language provides a calling mechanism for host language programs to execute SQL statements contained in a separate file called an SQL module file. The file contains module language elements that specify a single SQL module. The module includes one or more procedures. A procedure can contain a:

- Simple statement, which consists of a single SQL statement and optional parameter declarations
- Compound statement, which can include local variable declarations, multiple SQL statements, flow control statements, and transaction management statements

A procedure that contains a single SQL statement is called a **simple-statement procedure**. A procedure that contains a compound statement, which can contain multiple SQL statements, is called a **multistatement procedure**.

The host language program uses call statements to specify a particular SQL module procedure and supplies a sequence of actual parameters that corresponds in number and in data type to the parameter declarations in the procedure. A call to a procedure in an SQL module causes the simple or compound statement in the procedure to be executed.

Oracle Rdb recommends using SQL module language, rather than precompiled SQL, because module language offers the following advantages:

- Module language allows procedures that contain SQL statements to be called from any host language. In contrast, the SQL precompiler only supports a subset of host languages: Ada, C, COBOL, FORTRAN, Pascal, and PL/I.
- Programs that use the SQL module language can isolate all SQL statements in SQL modules to improve modularity and avoid using two languages in the same source file.
- Programs can work around restrictions of the SQL precompiler by calling SQL modules:
  - Programs that support pointer variables can take full advantage of dynamic SQL and use the SQLDA and SQLDA2 with the SQL module language.
  - SQL module language does not restrict use of host language features not supported by the precompiler (such as pointer variables in C, block structure, macros, user-defined types, and references to array elements).
- Programs written in languages for which there is an ANSI standard can avoid embedding code that does not conform to the standard by isolating noncompliant SQL statements in SQL modules.

For a detailed discussion of programming considerations for the SQL module language, see the *Oracle Rdb Guide to SQL Programming*.

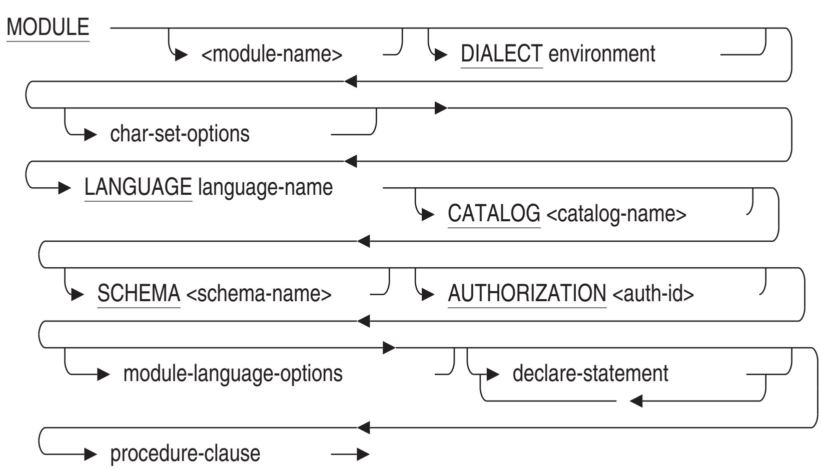
3.2 SQL Module Language Syntax

The SQL module language provides special keywords and syntax allowing procedures containing SQL statements to be called from host languages that are not supported by the SQL precompiler.

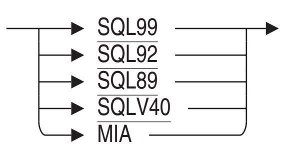
Environment

SQL module language elements must be part of an SQL module file.

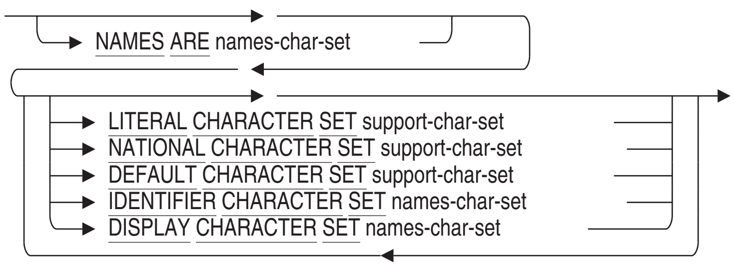
Format



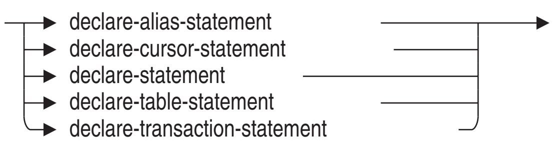
environment =



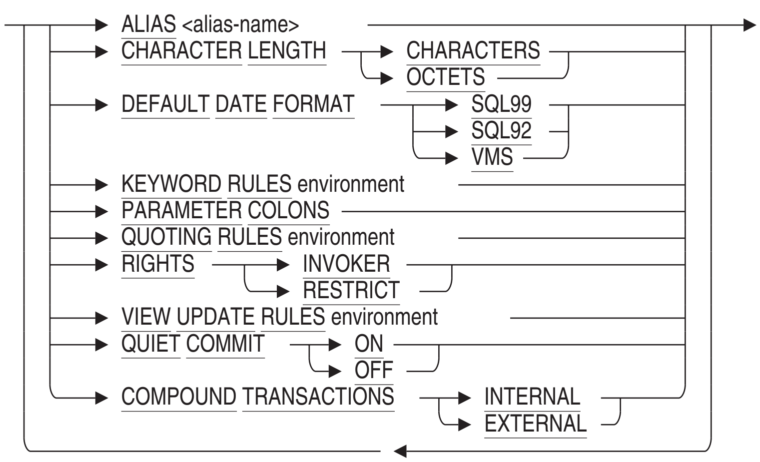
char-set-options =



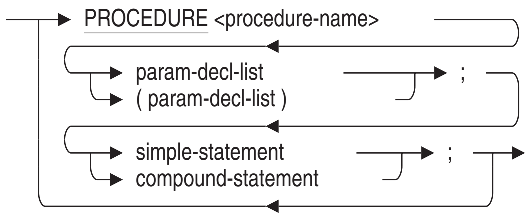
declare-statement =



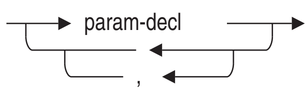
module-language-options =



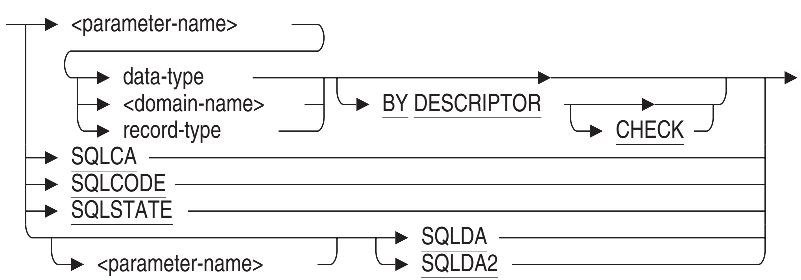
procedure-clause =



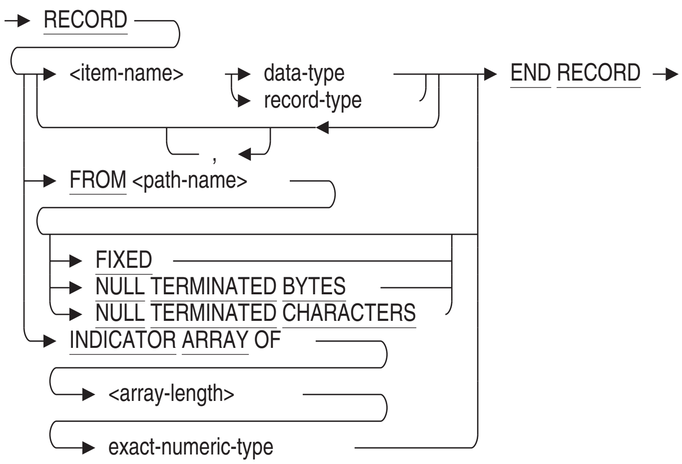
param-decl-list =



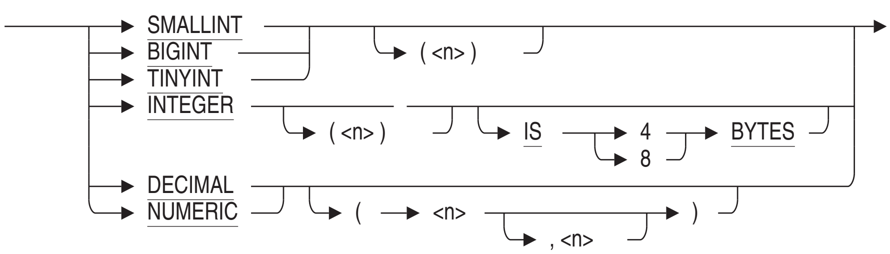
param-decl =



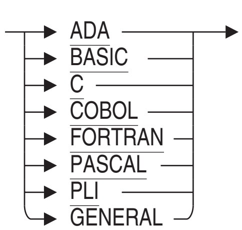
record-type =



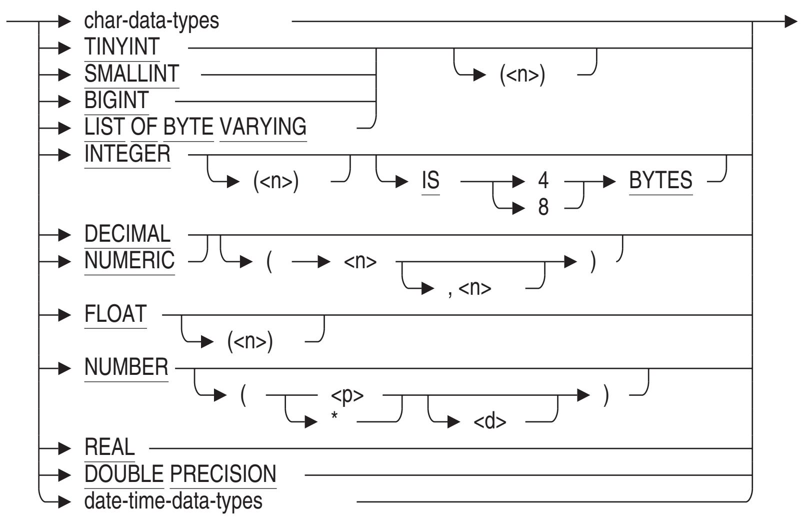
exact-numeric-type =



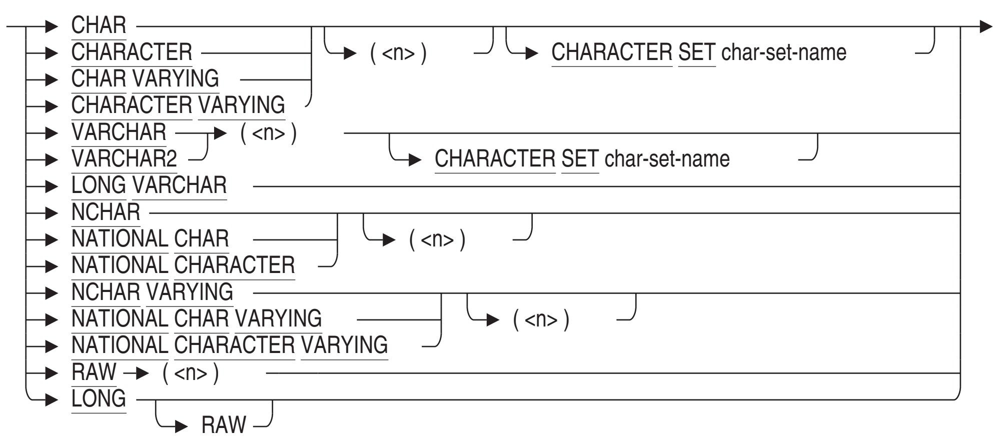
language-name =



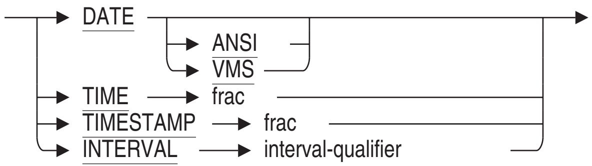
data-type =



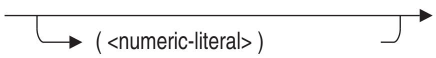
char-data-types =



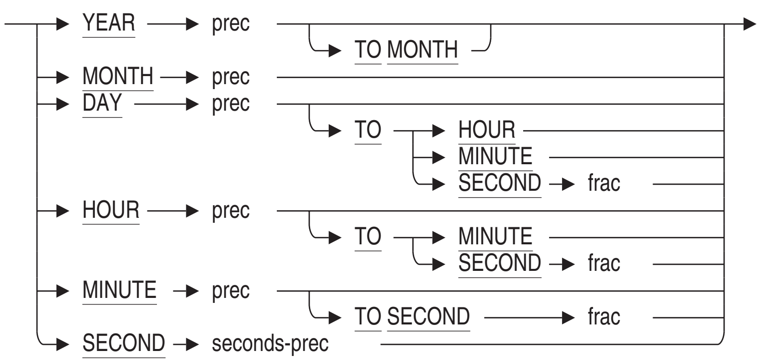
date-time-data-types =



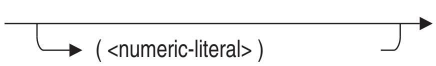
frac =



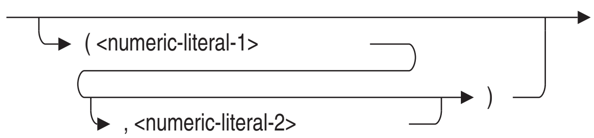
interval-qualifier =



prec =



seconds-prec =



Arguments

ALIAS alias-name
Specifies the default alias for the module. If you do not specify a module alias, the default alias is the authorization identifier for the module.

When the FIPS flagger is enabled, the ALIAS clause (by itself or used with the AUTHORIZATION clause) is flagged as nonstandard syntax.

If the application needs to refer to only one database across multiple modules, it is good practice to use the same alias for the default database in all modules that will be linked to make up an executable image. If that image will include modules processed with the SQL precompiler, you should specify RDB$DBHANDLE in the AUTHORIZATION clause of all SQL modules in the image because the alias RDB$DBHANDLE always designates the default database in precompiled SQL programs.

AUTHORIZATION auth-id
Specifies the authorization identifier for the module. If you do not specify a schema authorization, the authorization identifier is the user name of the user compiling the module.

If you want to comply with the ANSI/ISO SQL89 standard, specify the AUTHORIZATION clause without the schema-name. Specify both the AUTHORIZATION clause and the schema name to comply with the ANSI/ISO SQL99 Standard.

When you attach to a multischema database, the authorization identifier for each schema is the user name of the user compiling the module. This authorization identifier defines the default alias and schema. You can use the ALIAS and SCHEMA clauses to override the defaults.

If you attach to a single-schema database or specify that MULTISCHEMA IS OFF in your ATTACH or DECLARE ALIAS statements and you specify both an AUTHORIZATION clause and an ALIAS clause, the authorization identifier is ignored by SQL unless you specify the RIGHTS clause in the module file. The RIGHTS clause causes SQL to use the authorization identifier specified in the module AUTHORIZATION clause for privilege checking. Refer to the description of the RIGHTS clause later in this section.

If procedures in the SQL module always qualify table names with an authorization identifier, the AUTHORIZATION clause has no effect on SQL statements in the procedures.

When the FIPS flagger is enabled, the omission of an AUTHORIZATION clause is flagged as nonstandard ANSI syntax.

BY DESCRIPTOR
Specifies that the formal parameter will be passed to the calling program module by descriptor. The BY DESCRIPTOR clause is useful when:

- You specify the GENERAL keyword in the LANGUAGE clause of an SQL module, but the default for the language is to pass parameters by descriptor. The default for GENERAL is to pass parameters by reference, but you can override that default passing mechanism by specifying BY DESCRIPTOR.
- You want to take advantage of the CHECK option for parameter declarations. That option is available only for parameters declared with the BY DESCRIPTOR clause.
- You need to override the default parameter passing mechanism for languages that pass parameters by reference.

The BY DESCRIPTOR clause supports only OpenVMS static descriptors, which are fixed-length fields.

For any language, the passing mechanism for SQL module formal parameters must be the same as the actual parameters in the host language module.

Ada, BASIC, C, FORTRAN, Pascal, and PL/I do not support passing records by descriptor. You may construct a descriptor from elements in all these languages and pass the constructed descriptor to the SQL module language by reference.

- When you construct a descriptor for a host language record when the module language is Ada, BASIC, C, FORTRAN, Pascal, PL/I, or GENERAL, use a fixed-length descriptor (CLASS_S) with a character string data type, and pass the length of the entire record.
- If the language is Ada, BASIC, FORTRAN, or Pascal, pass indicator arrays using an array descriptor (CLASS_A) and the data type of all of the array elements.

- If the language is COBOL, pass arrays using fixed-length (CLASS_S) descriptors and character string data types, regardless of the data types of the array elements.
- If the language is C, the SQL module processor interprets CHAR fields one way when the data type is defined in the module, and another way when the definition is read from the dictionary. When the data type is defined in the module, the SQL module processor interprets character strings within records as null-terminated strings. In other words, if you declare a field specified as CHAR(9), the C module language interprets this as a field that is actually 10 characters long, with the tenth character being the null terminator.

  However, if you include a record in a C module from the data dictionary, you can specify any of three options for CHAR field interpretation. For details, see FIXED, NULL TERMINATED BYTES, and NULL TERMINATED CHARACTERS in the Arguments section.

CATALOG catalog-name
Specifies the default catalog for the module. Catalogs are groups of schemas within a multischema database. If you omit the catalog name when specifying an object in a multischema database, SQL uses the default catalog name RDB$CATALOG. Databases created without the multischema attribute do not have catalogs. You can use the SET CATALOG statement to change the current default catalog name in dynamic or interactive SQL.

CHARACTER LENGTH
Specifies whether the length of character string parameters, columns, and domains are interpreted as characters or octets. If the dialect is set to SQL89, SQL92, SQL99 or MIA, the default is CHARACTERS. Otherwise, the default is OCTETS.

char-data-types
Refer to Section 2.3 for information about the character data types that SQL supports.

CHECK
Specifies that SQL compares at run time the data type, length, and scale of the descriptor for an actual parameter to what was declared for the procedure parameter in the SQL module. If the two do not match, SQL returns an error. The CHECK clause works only with parameters passed by descriptor from the calling host language module.

Because there is no connection between an SQL module and a calling host language program module when they are compiled, there is no way for SQL to check for agreement between formal parameter declarations and actual parameters in calls to the module. The CHECK clause provides a way to do such checking when the program runs.

If a formal parameter declaration does not specify the CHECK clause, SQL assumes that procedure and calling parameters agree. If they do not, programs can give unpredictable results. However, you may choose not to use the CHECK clause because:

- The CHECK clause is not part of ANSI-standard SQL.
- There is a minor performance penalty for SQL to check parameters at run time.
- Using CHECK can make host programs more complicated.

The CHECK clause follows these rules in comparing formal parameters with call parameters:

- If a formal parameter is TIMESTAMP data type, the CHECK clause accepts any corresponding actual parameter that is 8 bytes long.
- If the language is C and the formal parameter is CHAR data type, the CHECK clause expects the descriptor to be 1 byte longer than the number of characters in the formal parameter. This occurs because character strings in C include a terminator character (they are in ASCIZ format) that is not included in the length of the formal parameter declaration.

  When you retrieve data definitions from the dictionary, however, you can change the default interpretation of character data by specifying FIXED or NULL TERMINATED CHARACTERS in the record-type clause of the FROM path-name clause.
- The CHECK clause supports dynamic string descriptors (CLASS_D) in BASIC for procedure parameters declared with the CHARACTER data type. However, the CHECK clause does not compare the length of the descriptor with the length of the procedure parameter because the buffer to receive the data is allocated at run time.
- If the formal parameter is VARCHAR data type, the descriptor that the CHECK clause accepts depends on the language.
  - If the language is PL/I or Pascal (languages that support varying character data type), the descriptor must be a varying string (CLASS_VS) descriptor, the data type must be varying text, and the length must be the same as the length of the formal parameter declaration.

– If the language is not PL/I or Pascal, the CHECK clause accepts a varying string descriptor as in the preceding paragraph, or a fixed-length (CLASS_S) or unspecified (DTYPE_Z) descriptor with data type of text and a length 2 bytes longer than the length of the formal parameter declaration.

For more detail on the different types of OpenVMS argument descriptors, see the OpenVMS programming documentation.

compound-statement

Most commonly, includes multiple executable SQL statements, associated variable declarations, and control statements within a BEGIN . . . END block; however, each of these arguments is optional. For instance, you can create an empty BEGIN . . . END block (BEGIN END;).

SQL executes the compound statement when the procedure in which it is embedded is called by a host language module. See the Compound Statement for more complete information about a compound statement.

COMPOUND TRANSACTIONS INTERNAL
COMPOUND TRANSACTIONS EXTERNAL

Allows you to specify whether SQL should start a transaction before executing a compound statement or stored procedure.

The COMPOUND TRANSACTIONS EXTERNAL clause instructs SQL to start a transaction before executing a procedure. The COMPOUND TRANSACTIONS INTERNAL clause instructs SQL to allow a procedure to start a transaction as required by the procedure execution.

By default, SQL starts a transaction before executing a compound statement if there is no current transaction

data-type

You can specify the character set of parameters that are defined as character data types. SQL assumes the character set of parameters based on the following rules:

- If a parameter is not qualified by a character set or defined as a national character data type, SQL considers the parameter to be of the default character set as specified in the DEFAULT CHARACTER SET clause.
- If a parameter is defined as a national character data type (NCHAR, NCHAR VARYING), SQL considers the parameter to be of the national character set as specified in the NATIONAL CHARACTER SET clause.

- If a parameter is defined as a data type qualified by a character set, SQL considers the parameter to be of that character set.

With the exception of the INTEGER data type, see Section 2.3 and Section 3.5 for information about data types and qualifying a data type with a character set. The following argument describes the INTEGER data type with regard to the SQL module language.

The SQL data type specified for the formal parameter in a module must be equivalent to the data type of the host language variable declaration for the actual parameter. If the formal parameter and actual parameter are not declared with equivalent data types, SQL can give unpredictable results. Section 3.5 shows which host language data types are equivalent to SQL data types and describes how to convert data types in a module procedure when there is no equivalent host language data type.

The data type for a database key is CHAR(n), where *n* equals the number of bytes of the database key. See Section 2.6.5 for more information on database keys.

date-time-data-types
frac
interval-qualifier
prec
seconds-prec
For information about specific data types and their qualifiers, see Section 2.3.

declare-statement
Any of the following statements:

- DECLARE ALIAS
- DECLARE CURSOR
- DECLARE STATEMENT
- DECLARE TABLE
- DECLARE TRANSACTION

You must place all DECLARE statements in an SQL module together after the LANGUAGE clause of the module. All such DECLARE statements are optional.

All the DECLARE statements except DECLARE TRANSACTION can be repeated. For each DECLARE CURSOR statement, however, there must be only one procedure in the SQL module that contains an OPEN statement that corresponds to the DECLARE CURSOR statement.

Do not use any punctuation to separate DECLARE statements or to separate the declare-statement section from the procedure section.

DEFAULT CHARACTER SET support-char-set

Specifies the character set for parameters that are not qualified by a character set and are not defined as a national character data type. If you do not specify a character set in this clause or in the NAMES ARE clause, the default is DEC_MCS. This clause overrides the character set specified in the NAMES ARE clause. See Section 2.1 for a list of the allowable character sets.

DEFAULT DATE FORMAT

Controls the default interpretation for columns with the DATE or CURRENT_TIMESTAMP data type. The DATE and CURRENT_TIMESTAMP data types can be either VMS or SQL format.

If you specify VMS, both data types are interpreted as VMS format. The VMS format DATE and CURRENT_TIMESTAMP contain YEAR to SECOND fields, like a TIMESTAMP.

If you specify an SQL standard such as SQL99, both data types are interpreted as SQL format. The SQL format DATE contains only the YEAR to DAY fields.

The default is VMS.

Use the DEFAULT DATE FORMAT clause, rather than the ANSI_DATE qualifier, because the qualifier will be deprecated in a future release.

DIALECT

Controls the following settings for the current connection:

- Whether the length of character string parameters, columns, and domains are interpreted as characters or octets
- Whether double quotation marks are interpreted as string literals or delimited identifiers
- Whether or not identifiers may be keywords
- Which views are read-only
- Whether columns with the DATE or CURRENT_TIMESTAMP data type are interpreted as VMS or SQL99 format
- Whether or not parameter names begin with a colon
- Whether or not the session character sets change depending on the dialect specified

The DIALECT clause lets you specify the settings with one clause, instead of specifying each setting individually. Because the module processor processes the module clauses sequentially, the DIALECT clause can override the settings of clauses (for example, QUOTING RULES) specified before it or be overridden by clauses specified after it.

The following statements are specific to the SQL92 and SQL99 dialects:

- The default constraint evaluation time setting changes from DEFERRABLE to NOT DEFERRABLE.
- Conversions between character data types when storing data or retrieving data raise exceptions or warnings in certain situations.
- 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 non-null terminated C strings.

See the SET DIALECT Statement for more information on the settings for each option of the DIALECT clause.

DISPLAY CHARACTER SET names-char-set
Specifies the character set used for automatic translation between applications and SQL. If you do not specify a character set the default is DEC_MCS. See Section 2.1.5 for a list of allowable character sets.

domain-name
You can specify an SQL data type directly or name a domain. If you name a domain, the parameter inherits the data type of the domain.

FIXED
The FIXED, NULL TERMINATED BYTES, and NULL TERMINATED CHARACTERS clauses tell the module processor how to interpret C language text fields. Example 3 in the Usage Notes section shows how the size of the text field you declare varies according to which of the three interpretation options you select.

If you specify FIXED, the module processor interprets CHAR fields from the dictionary as fixed-length character strings.

FROM path-name
Specifies the data dictionary path name of a data dictionary record definition. You can use this clause to retrieve data definitions from the dictionary.

The data dictionary record definition that you specify cannot contain any OCCURS clauses or arrays. You must specify a data dictionary record definition that contains only valid SQL or Oracle Rdb data types.

The FROM path-name clause cannot be used in a second-level record specification (a record-type that you specify within record-type).

IDENTIFIER CHARACTER SET names-char-set
Specifies the character set used for object names such as cursor names and table names. If you do not specify a character set in this clause or in the NAMES ARE clause, the default is DEC_MCS. This clause overrides the character set specified in the NAMES ARE clause. See Section 2.1.5 for a list of allowable character sets.

The specified character set must contain ASCII.

Note

If the dialect or character sets are not specified in the module header, SQL uses the RDB$CHARACTER_SET logical name to determine the character sets to be used by the database. See Section 2.1.5 and Appendix E for more detail regarding the RDB$CHARACTER_SET logical name.

The RDB$CHARACTER_SET logical name is deprecated and will not be supported in a future release.

INDICATOR ARRAY OF
Specifies a one-dimensional array of elements with one of the data types shown in the exact-numeric-type diagram. An indicator array provides indicator parameters for fields in the host structure. The indicator array must have at least as many elements in it as the record definition has.

You cannot use an indicator array as a record or contain it within a record. In other words, the INDICATOR ARRAY OF clause cannot be used in a second-level record specification (a record-type that you specify within record-type).

You cannot explicitly refer to individual elements in an indicator array. For this reason, you cannot use indicator arrays in UPDATE statements or WHERE clauses.

For more information about indicator arrays, see Section 2.2.13.2.

item-name

Specifies the name of an item in a record. Do not give the same name for two record items at the same level in the same record declaration.

When SQL statements within a procedure refer to an item name within a subrecord in the same procedure as a parameter declaration, they must fully qualify the item name with the record name and all intervening subrecord names. Separate record names from item names with periods.

KEYWORD RULES

Controls whether or not identifiers can be keywords. If you specify SQL92, SQL99, SQL89, or MIA, you cannot use keywords as identifiers, unless you enclose them in double quotation marks. If you specify SQLV40, you can use keywords as identifiers. The default is SQLV40.

Use the KEYWORD RULES clause, rather than the ANSI_IDENTIFIER qualifier, because the qualifier will be deprecated in a future release.

LANGUAGE language-name

A keyword that specifies the name of the host language in which the program is written. This program calls the procedures in the module. Specify GENERAL for languages that do not have a corresponding keyword in the LANGUAGE clause.

The language identifier determines:

- The kinds of data types that the SQL module processor considers valid in the module's formal parameter declarations. If a language does not support a data type equivalent to some SQL data type, the SQL module processor generates a warning message when it encounters the data type in a formal parameter. (A formal parameter is the name in an SQL module procedure declaration that represents the corresponding actual parameter in a host language call to the SQL module procedure.)

  For example, SQL supports the BIGINT data type, but PL/I does not. The module processor generates a warning message when it encounters a BIGINT formal parameter in an SQL module that specifies the PL/I language in the LANGUAGE section.

- The default mechanism for passing parameters to and from a host language source file. Parameters are always passed by the default passing mechanism for the language specified in the language clause. Table 3–1 shows those defaults.

Table 3–1 Default Passing Mechanism for Host Languages to SQL Modules

Language	Passing Mechanism
Ada	By reference
BASIC	CHAR by descriptor; all others by reference
C	By reference
COBOL	By reference
FORTRAN	CHAR, SQLCA, SQLDA by descriptor; all others by reference
Pascal	By reference
PL/I	By reference
GENERAL	By reference

- The default data type that SQL expects for certain actual parameters.

  In COBOL, for example, if a DOUBLE PRECISION formal parameter is declared in an SQL module procedure, the procedure expects the parameter to be passed from the calling module as D_FLOAT rather than G_FLOAT because COBOL does not support G_FLOAT. Similarly, in C, if a CHAR(n) formal parameter is declared in an SQL module procedure, the procedure expects the parameter to be passed from the calling module as an ASCIZ string with a length of (n+1).

LITERAL CHARACTER SET support-char-set
Specifies the character set for literals that are not qualified by a character set or national character set. If you do not specify a character set in this clause or in the NAMES ARE clause, the default is DEC_MCS. This clause overrides the character set for unqualified literals specified in the NAMES ARE clause. See Section 2.1 for a list of the allowable character sets.

MODULE module-name
An optional name for the module. If you do not supply a module name, the default name is SQL_MODULE.

Use any valid operating system name. (See Section 2.2 for more information on user-supplied names.) However, the name must be unique among the modules that are linked together to form an executable image.

NAMES ARE names-char-set

Specifies the character set used for the default, identifier, and literal character sets for the module. This clause also specifies the character string parameters that are not qualified by a character set or national character set. If you do not specify a character set, the default is DEC_MCS.

The character set specified in this clause must contain ASCII. See Section 2.1.5 for a list of the allowable character sets.

NATIONAL CHARACTER SET support-char-set

Specifies the character set for literals qualified by the national character set and for parameters defined as a national character data type (NCHAR, NCHAR VARYING). If you do not specify a character set in this clause, the default is DEC_MCS. See Section 2.1 for a list of the allowable character sets.

NULL TERMINATED BYTES

Specifies that text fields from the dictionary are null-terminated. The module processor interprets the length field in the dictionary as the number of bytes in the string. If n is the length in the dictionary, then the number of data bytes is $n - 1$ and the length of the string is n bytes.

In other words, the module processor assumes that the last character of the string is for the null terminator. Thus, a field that the dictionary lists as 10 characters can hold only a 9-character SQL field from the C module language. (Other module languages could fit a 10-character SQL field into it.)

If you do not specify a character interpretation option, NULL TERMINATED BYTES is the default.

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.

parameter-name

The name for a formal parameter. Use any valid SQL name. See Section 2.2 for more information on user-supplied names.

Formal parameter names do not have to be the same as the host language variables for the actual parameters to which they correspond. However, making the names the same is a useful convention for keeping track of which parameter corresponds to which host language variable.

SQLCA, SQLCODE, SQLDA, SQLDA2, and SQLSTATE are special-purpose parameters and do not require user-supplied names (although you can optionally specify a parameter name with SQLDA or SQLDA2).

There are three ways to specify a valid SQL data type for the formal parameter:

- data-type
- domain-name
- record-type

PARAMETER COLONS

If you use the PARAMETER COLONS clause, all parameter names must begin with a colon (:). This rule applies to both declarations and references of module language procedure parameters. If you do not use this clause, no parameter name can begin with a colon.

The current default behavior is no colons are used. However, this default is deprecated syntax. In the future, colons will be the default because it allows processing of ANSI-standard modules.

Use the PARAMETER COLONS clause, rather than the ANSI_PARAMETERS qualifier, because the qualifier will be deprecated in a future release.

PROCEDURE procedure-name

Specifies the name of a procedure. Use any valid OpenVMS name.) (See Section 2.2 for more information on user-supplied names.)

The procedure name is used in host language calls to specify a particular procedure. In addition to a procedure name, a procedure in an SQL module must contain one or more parameter declarations and an SQL statement.

QUIET COMMIT ON
QUIET COMMIT OFF

The QUIET COMMIT ON clause disables error reporting for the COMMIT and ROLLBACK statements if either statement is executed when no transaction is active. The QUIET COMMIT OFF clause enables error reporting for the COMMIT and ROLLBACK statements if either statement is executed when no transaction is active:

```
MODULE TXN_CONTROL
LANGUAGE BASIC
PARAMETER COLONS
QUIET COMMIT ON
PROCEDURE S_TXN (SQLCODE);
SET TRANSACTION READ WRITE;
```

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

The QUIET COMMIT OFF clause is the default.

QUOTING RULES

Controls whether double quotation marks are interpreted as string literals or delimited identifiers. If you specify SQL99, SQL92, SQL89, or MIA, SQL interprets double quotation marks as delimited identifiers. If you specify SQLV40, SQL interprets double quotation marks as literals. The default is SQLV40.

Use the QUOTING RULES clause, rather than the ANSI_QUOTING qualifier, because the qualifier will be deprecated in a future release.

RECORD . . . END RECORD

Specifies the beginning and end of the record that you are supplying in a module language parameter declaration.

A record definition cannot contain an SQLDA, an SQLDA2, an SQLCODE, an SQLCA, or an SQLSTATE.

record-type

You can pass records and indicator arrays to SQL module language procedures using the record-type clause.

You can also pass records and indicator arrays to SQL module language procedures and retrieve data dictionary record declarations using the record-type clause.

If a record reference has an indicator, it must be an indicator array. Specify the INDICATOR ARRAY OF clause instead of an item name or path name.

The following example shows the use of record structures and indicator arrays in an SQL module language program. Because parameters in the module are preceded by colons, you must include the PARAMETER COLONS clause in the module header.

```
MODULE          employee_module
LANGUAGE        pascal
AUTHORIZATION   pers
PARAMETER COLONS

DECLARE pers ALIAS FOR FILENAME mf_personnel

DECLARE WORK_STATUS_CURSOR CURSOR FOR
    SELECT *
    FROM   PERS.WORK_STATUS
```

```
PROCEDURE OPEN_WORK_STATUS
    SQLCODE;

    OPEN WORK_STATUS_CURSOR;

PROCEDURE CLOSE_WORK_STATUS
    SQLCODE;

    CLOSE WORK_STATUS_CURSOR;

PROCEDURE FETCH_EMPS_TO_DEPS_CURSOR
    SQLCODE,
    :work_status_rec
        record
        status_code PERS.work_status.STATUS_CODE_DOM
        status_name PERS.work_status.STATUS_NAME_DOM
        status_type PERS.work_status.STATUS_DESC_DOM
        end record
    :ind_array
        record
        indicator array of 3 SMALLINT
        end record
    ;
    FETCH WORK_STATUS_CURSOR
    INTO :work_status_rec INDICATOR :ind_array;
```

RIGHTS

Specifies whether or not a module must be executed by a user whose authorization identifier matches the module authorization identifier.

If you specify RESTRICT, SQL bases privilege checking on the default authorization identifier. The default authorization identifier is the authorization identifier of the user who compiles a module unless you specify a different authorization identifier using an AUTHORIZATION clause in the module. The RESTRICT option causes SQL to compare the user name of the person who executes a module with the default authorization identifier and prevent any user other than one with the correct authorization identifier from invoking that module. All applications that use multischema will be the invoker by default.

If you specify INVOKER, SQL bases the privilege on the authorization identifier of the user running the module.

The default is INVOKER.

Use the RIGHTS clause, rather than the ANSI_AUTHORIZATION qualifier, because the qualifier will be deprecated in a future release.

SCHEMA schema-name

Specifies the default schema name for the module. The **default schema** is the schema to which SQL statements refer if those statements do not qualify table and other schema names with an authorization identifier. If you do not specify a default schema name for a module, the default schema name is the same as the authorization identifier.

Using the SCHEMA clause, separate SQL modules can each declare different schemas as default schemas. This can be convenient for an application that needs to refer to more than one schema. By putting SQL statements that refer to a schema in the appropriate module's procedures, you can minimize tedious qualification of schema element names in those statements.

When you specify SCHEMA schema-name AUTHORIZATION authorization-name, you specify the schema name and the schema authorization identifier for the module. The schema authorization identifier is considered the owner and creator of the schema and everything in it.

When the FIPS flagger is enabled for entry-level SQL92 or lower, the SCHEMA clause (by itself or used with the AUTHORIZATION clause) is flagged as nonstandard ANSI syntax.

If procedures in the SQL module always qualify table names with an authorization identifier, the SCHEMA clause has no effect on SQL statements in the procedures.

SQLCA

A formal parameter for the SQLCA (see Appendix C for more information on the SQLCA). The calling program module must declare a record that corresponds to the structure of the SQLCA and specify that record declaration as the calling parameter for the SQLCA formal parameter. Appendix C.3 gives examples of record declarations for the SQLCA parameter for supported calling languages.

Specifying SQLCA as a formal parameter is an alternative to specifying SQLCODE. Using SQLCA instead of SQLCODE lets the calling program module take advantage of the information SQL puts in the third element of the SQLERRD array in the SQLCA. Future versions of SQL may use the SQLCA for additional information.

SQLCODE

A formal parameter that SQL uses to indicate the execution status of the SQL statement in the procedure. The SQLCODE formal parameter does not require a data type declaration; SQL automatically declares SQLCODE with an INTEGER data type. However, the calling program module must still declare an integer variable for the actual parameter that corresponds to SQLCODE. The SQLCODE parameter must be passed by reference.

Oracle Rdb recommends that you use the SQLSTATE status parameter rather than SQLCODE. SQLSTATE complies with ANSI/ISO SQL standard and SQLCODE may be deprecated in a future release of Oracle Rdb.

See Table C-1 for more information about SQLCODE.

SQLDA
SQLDA2
A formal parameter for the SQLDA or SQLDA2 (see Appendix D for more information on the SQLDA and SQLDA2). The calling program module must declare a record that corresponds to the structure of the SQLDA or SQLDA2 and specify that record declaration as the calling parameter for the SQLDA or SQLDA2 formal parameter. You can optionally precede SQLDA or SQLDA2 in the parameter declaration with another name the SQL statement in the module procedure can use to refer to the SQLDA or SQLDA2. Appendix D gives examples of record declarations for the SQLDA and SQLDA2 parameters for supported calling languages.

SQLSTATE
A formal parameter that SQL uses to indicate the execution status of the SQL statement in the procedure. The SQLSTATE formal parameter does not require a data type declaration; SQL automatically declares SQLSTATE with a CHAR(5) data type. However, the calling program module must still declare a character variable for the actual parameter that corresponds to SQLSTATE. The SQLSTATE parameter must be passed by reference.

Oracle Rdb recommends that you use the SQLSTATE status parameter rather than SQLCODE. SQLSTATE complies with the ANSI/ISO SQL standard and SQLCODE may be deprecated in a future release of Oracle Rdb.

VIEW UPDATE RULES
Specifies whether or not the SQL module processor applies the ANSI/ISO standard for updatable views to all views created during compilation.

If you specify SQL92, SQL99, SQL89, or MIA, the SQL module processor applies the ANSI/ISO standard for updatable views to all views created during compilation. Views that do not comply with the ANSI/ISO standard for updatable views cannot be updated. The default is SQLV40.

The ANSI/ISO 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 you specify SQLV40, SQL does not apply the ANSI/ISO standard for updatable views. 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 or a view that can be updated.
- The GROUP BY clause is not specified.
- The HAVING clause is not specified.

Usage Notes

- Procedures in an SQL module can be in any order. They do not have to correspond to the order in which they are called by a host language module.
- When you use the SQL module processor and specify the C module language, SQL translates all C character strings as null-terminated strings. This means that when SQL passes these character strings from the database to the program, it reserves space at the end of the string for the null character. When a program passes a character string to the database for input, SQL looks for the null character to determine how many characters to store in the database. SQL stores only those characters that precede the null character; it does not store the null character.

  Because of the way in which SQL translates C character strings, you may encounter problems with applications that pass binary data to and from the database. To avoid these problems when you use the SQL module language with a C host language program, specify the module language as GENERAL.

When you retrieve data definitions from the dictionary, you can change the default translation of character data by specifying a character interpretation option in the record-type clause in the FROM path-name clause. For more information, see the descriptions of FIXED, NULL TERMINATED BYTES, and NULL TERMINATED CHARACTERS in the Arguments section.

The way in which SQL translates C character strings also affects programs that use the SQL INCLUDE or the SQL FROM path-name clause to copy record definitions from a data dictionary.

- The double hyphen (--) specifies that all remaining text on a line is a comment. The SQL module processor therefore ignores any text to the right of a double hyphen when it processes source files. You can also use blank lines to make your SQL module source file easier to read and understand. In addition, you can specify a comment on the same line as, but to the right of, any code that the SQL module processor requires. For example:

  ```
  DECLARE VI_DB ALIAS
          FOR FILENAME personnel -- Declare the alias for the database.
  ```

- You cannot continue a keyword, user-defined name, or literal (such as a quoted string) from one line to the next in SQL modules. Completely enter any of these on one line of your SQL module source file.
- Programs that call SQL modules and need to use the SQLCA and message vector for error handling must declare those structures explicitly. For information on declaring the SQLCA and message vector, see Appendix C.
- You can use SQL error handling routines such as sql_signal and sql_get_error_text routines in module language programs. For more information, see the *Oracle Rdb Guide to SQL Programming*.
- You cannot specify a WHENEVER statement in an SQL module. Furthermore, you cannot embed a WHENEVER statement in a host language source file that will be precompiled and expect it to apply to your calls to SQL module procedures. The WHENEVER statement is supported only by the SQL precompiler, which can identify only SQL statements embedded in a host language source file.

  Instead of an embedded WHENEVER statement, use a host language conditional statement to evaluate the SQL statement status field (called SQLCODE in the SQLCA) or the SQLSTATE status parameter (ANSI/ISO SQL standard) immediately following the call. For general information on error handling in programs, see the chapter on handling run-time errors in the *Oracle Rdb Guide to SQL Programming*.

- A host language program module can refer to more than one SQL module in its calls.
- If a DECLARE TABLE statement appears before a CREATE DATABASE statement, your compilation could fail with an error message indicating that SQL$DATABASE or SQL_DATABASE could not be opened or that certain database objects could not be found in your database.

  The SQL module language compiler processes metadata statements before other statements. If your DECLARE TABLE statement is found before the CREATE DATABASE (or CREATE ALIAS) statement that defines it, then SQL will try to attach to SQL$DATABASE or SQL_DATABASE for the metadata lookups.

  Place your CREATE DATABASE or CREATE ALIAS statement before your DECLARE TABLE statements.

Examples

Example 1: Calling an SQL module procedure from a Pascal program

The following example is a Pascal program that calls a procedure in an SQL module file:

```
PROGRAM list_employees(OUTPUT);

{
  Program to list employees' names whose last name matches a LIKE
  predicate.
  Note the following:
    1) The input parameter (like_string) to the SELECT expression
       in the DECLARE CURSOR is supplied on the OPEN_CURSOR call.
    2) The output parameters are returned on each FETCH_INTO call.
    3) The cursor is closed after the desired rows are processed,
       so that it will be positioned properly in subsequent
       operations.
 }

TYPE
 LAST_NAME = PACKED ARRAY[1..14] OF CHAR;
 FIRST_NAME = PACKED ARRAY[1..10] OF CHAR;

VAR
       { Variable data }

 sqlcode : INTEGER := 0;
 emp_last : LAST_NAME;
 emp_first: FIRST_NAME;
 like_string : LAST_NAME := 'T_ _ _ _ _ _ _ _ _ _ _ _ _ _';

       { Declarations of entry points in the SQL module }
```

```
 PROCEDURE SET_TRANS (VAR sqlcode : INTEGER); EXTERNAL;
 PROCEDURE OPEN_CURSOR (VAR sqlcode: INTEGER;
                           name : LAST_NAME); EXTERNAL;
 PROCEDURE FETCH_INTO (VAR sqlcode : INTEGER;
                      VAR last : LAST_NAME;
                      VAR first : FIRST_NAME); EXTERNAL;
 PROCEDURE CLOSE_CURSOR (VAR sqlcode : INTEGER); EXTERNAL;
 PROCEDURE ROLLBACK_TRANS (VAR sqlcode : INTEGER); EXTERNAL;
BEGIN
 SET_TRANS (sqlcode);              { Start a read-only transaction.}
 OPEN_CURSOR (sqlcode, like_string);{ Open the cursor, supplying    }
                                   { the string to match against.  }
 WRITELN('Matching Employees:');   { Print header.                 }
 REPEAT                            { Iterate matching names.       }
  BEGIN
  FETCH_INTO (sqlcode, emp_last, emp_first);{ Fetch the next name.  }
  IF sqlcode = 0
  THEN
   WRITELN(emp_first, emp_last);   { Print employee information.   }
  END
 UNTIL sqlcode <> 0;
 IF sqlcode <> 100                 { Print any error information.  }
 THEN
  WRITELN ('SQL error code = ', sqlcode);
 CLOSE_CURSOR (sqlcode);           { Finish the cursor operation.  }
 ROLLBACK_TRANS (sqlcode);         { Finish the transaction.       }
END.
```

Here is the SQL module file that this program calls:

```
MODULE employees
LANGUAGE PASCAL
AUTHORIZATION SQL_USER
ALIAS RDB$DBHANDLE

 DECLARE ALIAS FOR FILENAME PERSONNEL

 DECLARE names CURSOR FOR
  SELECT LAST_NAME, FIRST_NAME
  FROM EMPLOYEES
  WHERE LAST_NAME LIKE match_string

 PROCEDURE SET_TRANS
  SQLCODE;
  SET TRANSACTION READ ONLY;

 PROCEDURE OPEN_CURSOR
  SQLCODE
  match_string CHAR(14);
  OPEN names;
```

```
PROCEDURE FETCH_INTO
 SQLCODE
 l_name CHAR(14)
 f_name CHAR(10);
 FETCH names INTO l_name, f_name;

PROCEDURE CLOSE_CURSOR
 SQLCODE;
 CLOSE names;

PROCEDURE ROLLBACK_TRANS
 SQLCODE;
 ROLLBACK;
```

Example 2: Calling an SQL module procedure from a C program

The following example is a C program that calls a procedure that is in an SQL module file:

```
/*
  C program to list employees' names where the last name matches a LIKE
  predicate.
  Note the following:
    1) The input parameter (like_string) to the SELECT expression
       in the DECLARE CURSOR is supplied on the OPEN_CURSOR call.
    2) The output parameters are returned on each FETCH_INTO call.
    3) The cursor is closed after the desired rows are processed,
       so that it will be positioned properly in subsequent operations.
 */

#include <stdio.h>

#pragma dictionary "name"

typedef struct name NAME_TYPE;
extern void FETCH_INTO (int *sqlcode, NAME_TYPE *name_record);

typedef char LAST_NAME[15];
typedef int *SQLCODE;

       /* Declarations of entry points in the SQL module */

extern void SET_TRANS (int *sqlcode);
extern void OPEN_CURSOR (int *sqlcode,
                         LAST_NAME name);


extern void CLOSE_CURSOR (int *sqlcode);
extern void ROLLBACK_TRANS (int *sqlcode);

void main ()
{
int sqlcode = 0;
NAME_TYPE name_record;
LAST_NAME like_string = "T%";

SET_TRANS (&sqlcode);                 /* Start a read-only transaction. */
if (sqlcode != 0)                     /* Print any error information. */
  printf ("SQL error code = %d\n", sqlcode);
```

```
OPEN_CURSOR (&sqlcode, like_string);  /* Open the cursor, supplying    */
                                      /* the string to match against.  */
if (sqlcode != 0)                     /* Print any error information. */
  printf ("SQL error code = %d\n", sqlcode);

printf ("Matching Employees:\n");     /* Print header. */
do                                    /* Iterate matching names. */
  {
  FETCH_INTO (&sqlcode, &name_record);/* Fetch the next name. */
  if (sqlcode == 0)
   printf ("%s%s\n", name_record.f_name, name_record.l_name);
  }                                   /* Print employee information. */
 while (sqlcode == 0);
if (sqlcode != 100)                   /* Print any error information. */
  printf ("SQL error code = %d\n", sqlcode);

CLOSE_CURSOR (&sqlcode);              /* Complete the cursor operation. */
if (sqlcode != 0)                     /* Print any error information. */
  printf ("SQL error code = %d\n", sqlcode);

ROLLBACK_TRANS (&sqlcode);            /* Finish the transaction. */
if (sqlcode != 0)                     /* Print any error information. */
  printf ("SQL error code = %d\n", sqlcode);

}
```

Here is the SQL module file that this program calls:

```
MODULE employees
LANGUAGE C
AUTHORIZATION SQL_USER
ALIAS RDB$DBHANDLE

DECLARE ALIAS FOR PATHNAME 'MF_PERSONNEL'

DECLARE names CURSOR FOR
 SELECT LAST_NAME, FIRST_NAME
 FROM EMPLOYEES
 WHERE LAST_NAME LIKE match_string


PROCEDURE SET_TRANS
 SQLCODE;
SET TRANSACTION READ ONLY;

PROCEDURE OPEN_CURSOR
 SQLCODE
 match_string CHAR(14);
OPEN names;

PROCEDURE FETCH_INTO
 SQLCODE,
 name_record RECORD FROM 'name' END RECORD;
FETCH names INTO name_record;

PROCEDURE CLOSE_CURSOR
 SQLCODE;
CLOSE names;
```

```
PROCEDURE ROLLBACK_TRANS
 SQLCODE;
ROLLBACK;
```

Here is a CDO command file that defines metadata used by the C program and SQL module. Field L_NAME has 15 characters, although the match_string in the SQL module file allows only 14 characters. The C programming language uses the character record and can only store a maximum of 14 characters plus the null terminator that C requires by default for character strings. See Example 3 for other character string interpretation options.

```
! MOD_LANG.CDO
!
! This file defines an Oracle CDD/Repository record to be used by
! the SQL module language module called from a C program.
!
DEFINE FIELD L_NAME DATATYPE TEXT 15.
DEFINE FIELD F_NAME DATATYPE TEXT 11.
DEFINE RECORD NAME.
L_NAME.
F_NAME.
END RECORD.
```

Example 3: Declaring text fields for the three different C language interpretation options

```
/*                  SQL$TEXT_FIELDS.C
*      This program demonstrates the use of DEC C and the SQL module language
*      to show different formats for text fields from the record PARTS,
*      stored in the repository.  The program tests each fetched field
*      to make sure that it ends in a null character if it is supposed to.
*
*      The program calls the SQL module SQL$TEXT_FIELDS_C.SQLMOD.
*      To create and populate the database for this example, you must run
*      the command procedure SQL$TEXT_FIELDS.SQL.  You must also have the
*      data dictionary installed on your system.
*
*/
#include stdio
main()
{
int sqlcode;
int i;
int fixed_okay;
```

```
/* Host variables for SQL calls.
*  Structure P_NTC shows the definition for a text
*  string interpreted with the NULL TERMINATED CHARACTERS option.
*  Character strings for P_NTC are 1 byte longer than those
*  character strings in the other three structures.
*  A field with a length of 7 bytes contains 6 characters
*  followed by the null value.
*/
struct
    { char pnum[7];
      char pname[21];
      char color[7];
      short weight;
      char city[16]; } p_ntc;
/*
*  Structure P_NTB shows the definition for a text
*  string interpreted with the NULL TERMINATED BYTES option.
*  A field with a length of 6 bytes contains 5 characters
*  followed by the null value.
*/
struct
    { char pnum[6];
      char pname[20];
      char color[6];
      short weight;
      char city[15]; } p_ntb;
/*
*  Structure P_DEFAULT shows the definition for a text
*  string interpreted without a character interpretation
*  option.  The default interpretation is the same as
*  NULL TERMINATED BYTES; a field with a length of 6 bytes
*  contains 5 characters followed by the null value.
*/
struct
    { char pnum[6];
      char pname[20];
      char color[6];
      short weight;
      char city[15]; } p_default;
/*
*  Structure P_FIXED shows the definition for a text
*  string interpreted with the FIXED option.
*  A field with a length of 6 bytes contains 6
*  characters.  There is no null value added to the field.
*/
struct
    { char pnum[6];
      char pname[20];
      char color[6];
      short weight;
      char city[15]; } p_fixed;
```

```
open_p( &sqlcode );
fetch_p_default( &sqlcode, &p_default );
close_p( &sqlcode );
printf( "%s, %s, %s, %s\n", p_default.pnum, p_default.pname,p_default.color,
       p_default.city );
for (i=0;i<6;i++) {
    if (p_default.pnum[i] == '\0') {
 if (i != 5) {
     printf("NULL not terminating in DEFAULT\n");
 } else {
     printf("DEFAULT is okay\n");
 }
    }
}
open_p( &sqlcode );
fetch_p_fixed( &sqlcode, &p_fixed );
close_p( &sqlcode );
printf( "%0.6s, %0.20s, %0.6s, %0.15s\n", p_fixed.pnum, p_fixed.pname,p_fixed.color,
       p_fixed.city );
fixed_okay = 1;
for (i=0;i<6;i++) {
    if (p_fixed.pnum[i] == '\0') {
 fixed_okay = 0;
    };
};
if (fixed_okay == 0) {
    printf("NULL in fixed string\n");
} else {
    printf("FIXED is okay\n");
};
open_p( &sqlcode );
fetch_p_ntb( &sqlcode, &p_ntb );
close_p( &sqlcode );
printf( "%s, %s, %s, %s\n", p_ntb.pnum, p_ntb.pname,p_ntb.color,
       p_ntb.city );
for (i=0;i<6;i++) {
    if (p_ntb.pnum[i] == '\0') {
 if (i != 5) {
     printf("NULL not terminating in NTB\n");
 } else {
     printf("NTB is okay\n");
 }
    }
}
```

```
open_p( &sqlcode );
fetch_p_ntc( &sqlcode, &p_ntc );
close_p( &sqlcode );
printf( "%s, %s, %s, %s\n", p_ntc.pnum, p_ntc.pname,p_ntc.color,
       p_ntc.city );
for (i=0;i<7;i++) {
    if (p_ntc.pnum[i] == '\0') {
 if (i != 6) {
     printf("NULL not terminating in NTC\n");
 } else {
     printf("NTC is okay\n");
 }
    }
}

}
```

Here is the SQL module file that this program calls:

```
-- This SQL module provides the SQL procedures needed by the
-- SQL$TEXT_FIELDS.C program.  The module illustrates the three
-- different ways that you can specify text fields in the
-- repository using the C programming language:
--        NULL TERMINATED BYTES, the default
--        NULL TERMINATED CHARACTERS
--        FIXED (no null)
--        FIXED (no null)
--
-- Because this module precedes parameter names with colons,
-- in compliance with the ANSI/ISO SQL standard, you must supply
-- the PARAMETER COLONS clause in the module header.
-------------------------------------------------------------------------------
-- Header Information Section
-------------------------------------------------------------------------------
MODULE SQL_TEXT_FIELDS_C
LANGUAGE C
AUTHORIZATION SQL_SAMPLE
PARAMETER COLONS
-------------------------------------------------------------------------------
-- DECLARE Statements Section
-------------------------------------------------------------------------------

DECLARE ALIAS FILENAME 'SUPPLIES'

DECLARE P_CURSOR CURSOR FOR SELECT * FROM PARTS

-------------------------------------------------------------------------------
-- Procedure Section
-- In every procedure, declare SQLCODE, a parameter that stores a value
-- representing the execution status of SQL statements.
-------------------------------------------------------------------------------

PROCEDURE open_p
    SQLCODE;
```

```
    OPEN P_CURSOR;

-- This procedure specifies the repository record PARTS using the
-- repository path name. Because none of the character interpretation
-- options is specified, output for a field defined as TEXT SIZE 6
-- in the repository or CHAR (6) in SQL will show the default interpretation,
-- NULL TERMINATED BYTES, a field of 6 bytes that contains 5
-- characters followed by a null value.

PROCEDURE fetch_p_default
    SQLCODE
    :P_REC  RECORD FROM 'CDD$DEFAULT.SUPPLIES.RDB$RELATIONS.PARTS' END RECORD;

    FETCH P_CURSOR INTO :P_REC;

-- This procedure specifies the repository record PARTS using the
-- repository path name. Because the FIXED option is specified,
-- output for a field defined as TEXT SIZE 6 in the repository or
-- CHAR (6) in SQL will be a field of 6 bytes that contains
-- 6 characters. There is no null value.

PROCEDURE fetch_p_fixed
    SQLCODE
    :P_REC  RECORD FROM 'CDD$DEFAULT.SUPPLIES.RDB$RELATIONS.PARTS'
     FIXED END RECORD;

    FETCH P_CURSOR INTO :P_REC;

-- This procedure specifies the repository record PARTS using the
-- repository path name. Because the NULL TERMINATED BYTES
-- option is specified, output for a field defined as TEXT SIZE 6
-- in the repository or CHAR (6) in SQL will be a field of 6 bytes
-- that contains 5 characters followed by the null value.

PROCEDURE fetch_p_ntb
    SQLCODE
    :P_REC  RECORD FROM 'CDD$DEFAULT.SUPPLIES.RDB$RELATIONS.PARTS'
     NULL TERMINATED BYTES END RECORD;

    FETCH P_CURSOR INTO :P_REC;

-- This procedure specifies the repository record PARTS using the
-- repository path name. Because the NULL TERMINATED CHARACTERS
-- option is specified, output for a field defined as TEXT SIZE 6
-- in the repository or CHAR (6) in SQL will be a field of 7
-- bytes that contains 6 characters followed by the null value.

PROCEDURE fetch_p_ntc
    SQLCODE
    :P_REC  RECORD FROM 'CDD$DEFAULT.SUPPLIES.RDB$RELATIONS.PARTS'
     NULL TERMINATED CHARACTERS END RECORD;

    FETCH P_CURSOR INTO :P_REC;

PROCEDURE close_p
    SQLCODE;
```

```
    CLOSE P_CURSOR;
```

Here is the SQL command procedure to create and populate the database used in these examples:

```
!
! This SQL procedure creates and populates the database used by
! the module language file SQL$TEXT_FIELDS_C.SQLMOD.
!
SET VERIFY
CREATE DATABASE FILENAME PERSONNEL PATHNAME 'CDD$TOP.PERSONNEL';
CREATE TABLE S (SNUM CHAR (5), SNAME CHAR (20), STATUS SMALLINT, CITY CHAR(15));
CREATE TABLE P (PNUM CHAR (6), PNAME CHAR(20),  COLOR CHAR(6),
  WEIGHT SMALLINT, CITY CHAR(15));
INSERT INTO P ( PNUM, PNAME, COLOR, WEIGHT, CITY )
VALUES ('P1', 'Nut', 'Red', 12, 'London' );
COMMIT;
DISCONNECT ALL;
```

3.3 Declaring the Length of Character Parameters

To ensure that you specify the length of character string parameters correctly, use the following guidelines:

- For C host language programs that call SQL modules declared with LANGUAGE C, any character parameters that correspond to character data type columns must be defined as the length of the longest valid column value in octets, plus 1 octet to allow for the null terminator.
- For other host language programs (or C host language programs that call SQL modules declared with LANGUAGE GENERAL), any character parameters that correspond to character data type columns must be defined as the length of the longest valid column value in octets.
- When calculating the length of the longest valid column value, you must take into consideration the number of octets for each character in the character set of the column and whether the SQL module language interprets the length of columns in characters or octets. A program can control how the SQL module language interprets the length of columns in the following ways:
  - The CHARACTER LENGTH clause of the module header or DECLARE MODULE statement
  - The DIALECT clause of the module header or DECLARE MODULE statement
  - For dynamic SQL, the SET CHARACTER LENGTH statement

  See Table 2–2 for information about the number of octets used for one character in each character set.

Assume that you create the database MIA_CHAR_SET with the following character sets:

- Default character set: DEC_KANJI
- National character set: KANJI
- Identifier character set: DEC_KANJI

Then, assume that the database contains the table COLOURS and that the columns in that table are defined as shown in the following example:

```
SQL> SHOW DOMAINS;
User domains in database with filename MIA_CHAR_SET
ARABIC_DOM                      CHAR(8)
         ISOLATINARABIC 8 Characters,  8 Octets
DEC_KANJI_DOM                   CHAR(16)
GREEK_DOM                       CHAR(8)
         ISOLATINGREEK 8 Characters,  8 Octets
HINDI_DOM                       CHAR(8)
         DEVANAGARI 8 Characters,  8 Octets
KANJI_DOM                       CHAR(8)
         KANJI 4 Characters,  8 Octets
KATAKANA_DOM                    CHAR(8)
         KATAKANA 8 Characters,  8 Octets
MCS_DOM                         CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
RUSSIAN_DOM                     CHAR(8)
         ISOLATINCYRILLIC 8 Characters,  8 Octets
SQL> --
SQL> SHOW TABLE (COLUMNS) COLOURS;
Information for table COLOURS

Columns for table COLOURS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
ENGLISH                         CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
FRENCH                          CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
JAPANESE                        CHAR(8)          KANJI_DOM
         KANJI 4 Characters,  8 Octets
ROMAJI                          CHAR(16)         DEC_KANJI_DOM
KATAKANA                        CHAR(8)          KATAKANA_DOM
         KATAKANA 8 Characters,  8 Octets
HINDI                           CHAR(8)          HINDI_DOM
         DEVANAGARI 8 Characters,  8 Octets
GREEK                           CHAR(8)          GREEK_DOM
         ISOLATINGREEK 8 Characters,  8 Octets
ARABIC                          CHAR(8)          ARABIC_DOM
         ISOLATINARABIC 8 Characters,  8 Octets
RUSSIAN                         CHAR(8)          RUSSIAN_DOM
         ISOLATINCYRILLIC 8 Characters,  8 Octets

SQL>
```

The following excerpt from an SQL module program shows how to specify the character sets, and how the character length is allocated in the module header:

```
--------------------------------------------------------------------------------
-- Header Information Section
--------------------------------------------------------------------------------
MODULE        SQL_MIA_CHAR_SET_C      -- Module name
DIALECT                 SQL92         -- Sets the character length to CHARACTERS
NAMES ARE               DEC_KANJI     -- Names character set
NATIONAL CHARACTER SET  KANJI         -- National character set
DEFAULT CHARACTER SET   DEC_KANJI     -- Default character set
LANGUAGE                C             -- Language of calling program
AUTHORIZATION           SQL_SAMPLE    -- Default authorization ID
ALIAS                   RDB$HANDLE    -- Default alias
   .
   .
   .
-- When you declare character string parameters, you must take into
-- account the character set of the corresponding SQL column, whether
-- the characters are single- or multiple-octet characters, and
-- whether the module specified the character length in octets
-- or characters.
--

-- Create domains

PROCEDURE CREATE_DOMAIN_MCS
    SQLCODE;
    CREATE DOMAIN MCS_DOM       CHAR (8) CHARACTER SET DEC_MCS;

--
-- The CREATE DATABASE statement and the module header identify
-- KANJI as the national character set. Thus, you can declare
-- the KANJI_DOM domain as data type NCHAR.
--

PROCEDURE CREATE_DOMAIN_KANJI
    SQLCODE;
    CREATE DOMAIN KANJI_DOM     NCHAR (4);

--
-- Because the module header and the CREATE DATABASE statement
-- define DEC_KANJI as the default character set, you do not
-- have to identify the character set for the DEC_KANJI_DOM
-- domain.
--

PROCEDURE CREATE_DOMAIN_DEC_KANJI
    SQLCODE;
    CREATE DOMAIN DEC_KANJI_DOM   CHAR (8);

PROCEDURE CREATE_DOMAIN_KATAKANA
    SQLCODE;
    CREATE DOMAIN KATAKANA_DOM  CHAR (8) CHARACTER SET KATAKANA;
```

.
.
.

You declare the corresponding parameters in the C host language program as shown in the following example:

```
/* When you declare character string parameters, you must take into    */
/* account the character set of the corresponding SQL column, whether  */
/* the characters are single- or multiple-octet characters, and        */
/* whether the module specifies the character length in octets         */
/* or characters.                                                       */

typedef char colour_string_t[17];

long    sqlcode;

enum languages
    {
    ENGLISH,
    FRENCH,
    JAPANESE,
    ROMAJI,
    KATAKANA,
    HINDI,
    GREEK,
    ARABIC,
    RUSSIAN,
    MAX_LANGUAGE
    };
enum colours
    {
    MAX_COLOUR = 6
    };

static char *language_name[] =  /* (NOTE: in the same sequence as the enum) */
    {
    "ENGLISH   ",
    "FRENCH    ",
    "JAPANESE  ",
    "ROMAJI    ",
    "KATAKANA  ",
    "HINDI     ",
    "GREEK     ",
    "ARABIC    ",
    "RUSSIAN   ",
    };

main()
{
static colour_string_t prism[MAX_LANGUAGE][MAX_COLOUR];
int     colour_count;
int     language;
int     colour;
```

```
CREATE_MIA_CHAR_SET_DB(&sqlcode);
if (sqlcode != SUCCESS)
    check_error();
.
.
.
```

3.4 Floating Point Number Representations

OpenVMS supports multiple representations for floating point numbers. These representations include the VAX F-Floating and IEEE S-Floating representations for single-precision (4 octet) numbers and the VAX G-Floating and D-Floating and the IEEE T-Floating representations for double-precision (8 octet) numbers. Modules generated by the SQL module language processor and by various host language processors support these representations for their single- and double-precision floating point data types. Host language modules which call SQL module language modules must have the floating point representations defined consistently with the procedure parameters of those modules or the values of the floating point numbers will not be correctly interpreted. This is true even if the host language and SQL data types are equivalent. For SQL module language, the floating point representation is determined by the value of the /FLOAT qualifier on the command line (see Section 3.6 for more details on the /FLOAT qualifier). For the host languages, it is determined by a combination of a /FLOAT or /[NO]G_FLOAT qualifier and /or internal language support for floating point representations (see Section 3.5 for details on matching host language floating point representations). The following code examples show how to ensure floating point representation consistency for various host languages using the IEEE formats.

In the discussion of actual parameter types below, examples will refer to the following SQL Module Language procedure which is assumed to yield a singleton select:

```
PROCEDURE GET_FLOATS (SQLCODE, REAL :P_FLOAT1, DOUBLE PRECISION :P_FLOAT2);
BEGIN
    SELECT MY_FLOAT1, MY_FLOAT2 INTO :P_FLOAT1, :P_FLOAT2 FROM A_TABLE
    WHERE KEY_VALUE = "1";
END;
```

The following example shows an Ada code fragment which is compatible with the GET_FLOATS sample procedure when the SQL Module Language program has been compiled with /FLOAT=IEEE_FLOAT:

```
procedure GET_FLOATS (
    SQLCODE : out INTEGER;
    P_FLOAT1 : out IEEE_SINGLE_FLOAT;
    P_FLOAT2: out IEEE_DOUBLE_FLOAT
    );
pragma INTERFACE (NONADA, GET_FLOATS);
SQLCODE : INTEGER;
FLOAT1 : SYSTEM.IEEE_SINGLE_FLOAT;
FLOAT2 : SYSTEM.IEEE_DOUBLE_FLOAT;
...
GET_FLOATS( SQLCODE, FLOAT1, FLOAT2);
```

BASIC provides a /REAL_SIZE qualifier which can be used to specify not only the size but the format of floating point variables declared using the REAL keyword. The relevant values for this qualifier for IEEE floating point formats are SFLOAT and TFLOAT. These values specify that REAL variables are to be of type S-Floating or T-Floating, respectively. BASIC also provides the OPTION command which allows the size and format of a REAL to be specified in a more local scope.

Additionally, BASIC has native datatypes (SFLOAT and TFLOAT) which explicitly specify S-Floating and T-Floating variables, respectively.

The following example shows a BASIC code fragment which is compatible with the GET_FLOATS sample procedure:

```
EXTERNAL GET_FLOATS(LONG, SFLOAT, TFLOAT)
DECLARE LONG SQLCODE
DECLARE SFLOAT FLOAT1
DECLARE TFLOAT FLOAT2
...
CALL GET_FLOATS(SQLCODE, FLOAT1, FLOAT2)
```

C has /FLOAT and /[NO]G_FLOAT qualifiers which work identically to those for SQL$MOD (except the default may be different). That is, the format of the floating point variables in the C program is determined by the qualifier. C has native types of "float" and "double" which are 32-bit and 64-bit floating point numbers, respectively. See Table 3–6 for more information on equivalency between SQL and C data types.

The following example shows a C code fragment which is compatible with the GET_FLOATS sample procedure provided that both the C module and the SQL Module Language program were compiled with the same setting of the /FLOAT or /[NO]G_FLOAT qualifier:

```
extern void GET_FLOATS (
    long *SQLCODE,
    float *P_FLOAT1,
    double *P_FLOAT2
);
long SQLCODE;
float float1;
double float2;
...
GET_FLOATS( &SQLCODE, &float1, &float2);
```

COBOL has a /FLOAT qualifier with the same options as SQL$MOD (except the default is D_FLOAT). There is no /[NO]G_FLOAT qualifer for COBOL. The /FLOAT qualifier works identically to that of SQL$MOD. That is, the format of the floating point variables in the COBOL program is determined by the qualifier. COBOL has native types of COMP-1 and COMP-2 which are 32-bit and 64-bit floating point numbers, respectively.

The following example shows a COBOL code fragment which is compatible with the GET_FLOATS sample procedure provided that both the COBOL program and the SQL Module Language program were compiled with the same setting of the /FLOAT qualifier:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      SQLCODE PIC S9(9)   USAGE COMP.
01      FLOAT1  COMP-1.
01      FLOAT2  COMP-2.
...
CALL "GET_FLOATS" USING SQLCODE, FLOAT1, FLOAT2.
```

FORTRAN has /FLOAT and /[NO]G_FLOAT qualifiers which work identically to those for SQL$MOD (except the default may be different). That is, the format of the floating point variables in the FORTRAN program is determined by the qualifier. FORTRAN has native types of "real" and "real*4" which are 32-bit floating point numbers and "double precision" and "real*8" which are 64-bit floating point numbers.

The following example shows a FORTRAN code fragment which is compatible with the GET_FLOATS sample procedure provided that both the FORTRAN module and the SQL Module Language program were compiled with the same setting of the /FLOAT or /[NO]G_FLOAT qualifier:

```
integer*4 SQLCODE
real*4 float1
real*8 float2
 . . .
CALL GET_FLOATS( SQLCODE, float1, float2)
```

Pascal has /FLOAT and /[NO]G_FLOAT qualifiers which work similarly to those for SQL$MOD (except the default may be different). That is, the format of floating point variables of certain data types in the Pascal program is determined by the qualifier. The Pascal native data types affected by the qualifiers are REAL, SINGLE and DOUBLE. The first two of these are 32-bit floating point numbers and the final one is a 64-bit floating point number. Pascal also has an Attribute called FLOAT which can be used to affect the format of floating point similarly to the /FLOAT qualifier but with a more local scope. Specifically, IEEE floating point format can be specified by using the IEEE_FLOAT keyword with the FLOAT Attribute.

In addition, Pascal has several format-specific floating point data types which specify a particular format regardless of the qualifier settings. The format-specific data types include S_FLOAT and T_FLOAT which are IEEE 32-bit and 64-bit floating point numbers, respectively.

The following example shows a Pascal code fragment which is compatible with the GET_FLOATS sample procedure when the SQL Module Language program was compiled with /FLOAT=IEEE_FLOATING:

```
sqlcode : INTEGER;
float1 : S_FLOAT;
float2 : T_FLOAT:
PROCEDURE GET_FLOATS
(   VAR SQLCODE     : INTEGER;
    VAR FLOAT_1     : S_FLOAT;
    VAR FLOAT_2     : T_FLOAT );
EXTERNAL;
 . . .
GET_FLOATS( sqlcode, float1, float2 );
```

PL/I has /FLOAT and /[NO]G_FLOAT qualifiers which work identically to those for SQL$MOD (except the default may be different). That is, the format of the floating point variables in the PLI program is determined by the qualifier. PL/I has a native type of FLOAT which can be a 32-bit and 64-bit floating point number depending on the size specification.

The following example shows a PL/I code fragment which is compatible with the GET_FLOATS sample procedure provided that both the PL/I module and the SQL Module Language program were compiled with the same setting of the /FLOAT or /[NO]G_FLOAT qualifier:

```
DECLARE GET_FLOATS EXTERNAL ENTRY (
        ANY REFERENCE, ANY REFERENCE, ANY REFERENCE);
DECLARE SFLOAT FLOAT(24) BINARY,
        TFLOAT FLOAT(53) BINARY,
        SQLCODE BIN FIXED(31);
 . . .
CALL GET_FLOATS( SQLCODE, SFLOAT, TFLOAT );
```

Note

Oracle Rdb always stores floating point numbers internally using the VAX 32-bit and 64-bit types called F-Floating (F_FLOAT) and G-Floating (G_FLOAT), respectively. This means that when IEEE formats are used in a host language program, Oracle Rdb converts back and forth between the VAX and IEEE formats. There are differences in the number of available bits in the fraction and exponent between these formats. Additionally, the IEEE formats have certain exponent values reserved for infinity values. These differences can cause floating point overflow or underflow as well as rounding errors during the conversion process. See Appendix A of the Portable Mathematics Library in the OpenVMS Operating System documentation for data on the maximum and minimum values for VAX versus IEEE floating point formats.

3.5 Equivalent SQL and Host Language Data Types

The SQL data type specified for the formal parameter in a module must be equivalent to the data type of the host language variable declaration for the actual parameter. If the formal parameter and actual parameter are not declared with equivalent data types, SQL can give unpredictable results. Refer to Table 3–3 through Table 3–10 to determine equivalent data types for each host language.

However, host languages typically do not support the same set of data types that SQL supports. To work with a column in a database defined with a data type not supported in a host language, the module must declare formal parameters of a data type that the host language supports. SQL automatically converts between the data type of the database column and the formal parameter when it processes the SQL statement in a procedure.

The following fragments from a BASIC program and its accompanying SQL module illustrate this technique. BASIC does not support a varying character string (VARCHAR) data type but does have a STRING data type that is equivalent to the SQL CHAR character string data type. The SQL module declares a formal parameter as CHAR and uses that parameter to pass values to and from a VARCHAR database column. SQL converts between the VARCHAR data type of the column and the CHAR data type of the formal parameter. The corresponding actual parameters in the BASIC calls to the SQL module are declared as STRING, which are compatible with the CHAR formal parameter.

Examples

Example 1: Inserting VARCHAR data with BASIC

```
   .
   .
   .
! Program declares STRING variables for use as actual parameters for
! procedures. VC_FIELD will contain values to be passed to and from a
! VARCHAR field in a table.

    DECLARE STRING EMPLOYEE_ID,VC_FIELD
   .
   .
   .
! Call an SQL module procedure that creates a table with a VARCHAR column:
```

```
    CALL CREATE_TABLE(sql_return_status)
    IF sql_return_status < 0 THEN
        CALL ROLLBACK_TRANSACTION(sql_return_status)
        PRINT 'Error creating table.  Exiting program.'
        EXIT PROGRAM
    END IF

! Call a procedure to insert a row into the table, using
! VC_FIELD to pass values to the VARCHAR column in the table.

    employee_id = '00550'
    vc_field = 'Inserting employee 550 into the table'

    CALL INSERT_VC(sql_return_status,employee_id,vc_field)
    IF sql_return_status < 0 THEN
        CALL ROLLBACK_TRANSACTION(sql_return_status)
        PRINT 'Error inserting row.  Exiting program.'
        EXIT PROGRAM
    END IF
```

Here is the corresponding SQL module fragment for that BASIC calling program:

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

-- This procedure creates the table with the VARCHAR column.

PROCEDURE CREATE_TABLE
    SQLCODE;

    CREATE TABLE VC_TABLE
        (
        EMPLOYEE_ID     CHAR(5),
        VC_FIELD        VARCHAR(80)
        );

-- This procedure inserts a row into the table. Note that the formal
-- parameter P_VC_FIELD is declared as CHAR to correspond with the actual
-- parameter, not as VARCHAR to correspond with its column in the table.

PROCEDURE INSERT_VC
    SQLCODE
    P_EMPLOYEE_ID       CHAR(5)
    P_VC_FIELD          CHAR(80);
```

```
    INSERT INTO VC_TABLE
        VALUES
            (
            P_EMPLOYEE_ID,
            P_VC_FIELD
            );
 .
 .
 .
```

An exception to this technique is required for SQL module procedures that need to handle table columns defined as one of the date-time data type. Because host languages do not support the date-time data types, calling programs that need to work with a table column defined as a date-time data type require special treatment.

Table 3–2 shows the OpenVMS data types that SQL requires for actual parameters when you declare formal parameters for each SQL data type.

Table 3–2 SQL and Corresponding OpenVMS Data Types for Module Language

Formal Parameter Data Type	Requires Actual Parameter of OpenVMS Data Type
CHAR (n)	Character string (DSC$K_DTYPE_T)
CHAR (n), qualified by character set	Character string (DSC$K_DTYPE_T)
NCHAR (n)	Character string (DSC$K_DTYPE_T)
VARCHAR (n)	Varying character string (DSC$K_DTYPE_VT)[1]
VARCHAR (n), qualified by character set	Varying character string (DSC$K_DTYPE_VT)[1]
NCHAR VARYING (n)	Varying character string (DSC$K_DTYPE_VT) [1]
LONG VARCHAR	Varying character string (DSC$K_DTYPE_VT)[1]
TINYINT [(n)][2]	Signed byte integer (DSC$K_DTYPE_B)
SMALLINT [(n)][2]	Signed word integer (DSC$K_DTYPE_W)
INTEGER [(n)][2]	Signed longword integer (DSC$K_DTYPE_L)

[1]Not supported in FORTRAN or BASIC; SQL generates a warning message.

[2]Scale factors not supported in C, FORTRAN, PL/I, Ada, Pascal, or BASIC; SQL generates a warning message.

(continued on next page)

Table 3–2 (Cont.) SQL and Corresponding OpenVMS Data Types for Module Language

Formal Parameter Data Type	Requires Actual Parameter of OpenVMS Data Type
BIGINT [(n)]	Signed quadword integer (DSC$K_DTYPE_Q)[1,4,5,9]
QUADWORD [(n)]	Signed quadword integer (DSC$K_DTYPE_Q)[1,4,5,9]
DECIMAL [(n)[,(n)]]	Packed decimal string (DSC$K_DTYPE_P)[1,3]
NUMERIC [(n)[,(n)]]	Numeric string, left separate sign (DSC$K_DTYPE_NL) [1,3,5]
FLOAT [(n)]	Single- or double-precision, floating-point number, depending on n. For single-precision: DSC$K_DTYPE_F or DSC$K_DTYPE_FS [6] and for double-precision: DSCK_DTYPE_G, DSCK_DTYPE_D, or DSC$K_DTYPE_FT. [7]
REAL	Single-precision, floating-point number (DSC$K_DTYPE_F or DSC$K_DTYPE_FS). [6]
DOUBLE PRECISION	Double-precision, floating-point number (DSCK_DTYPE_G, DSCK_DTYPE_D, or DSC$K_DTYPE_FT). [7]
(DATE)	No equivalent OpenVMS data type; two-longword array
DATE ANSI	No equivalent OpenVMS data type; two-longword array
DATE VMS	Absolute date and time (DSC$K_DTYPE_ADT)
TIME	No equivalent OpenVMS data type; two-longword array

[1]Not supported in FORTRAN or BASIC; SQL generates a warning message.

[3]Not supported in C, Ada, or Pascal; SQL generates a warning message.

[4]Not supported in PL/I; SQL generates a warning message.

[5]Not supported in BASIC; SQL generates a warning message.

[6]The floating point representation of single-precision numbers is determined by the value of the /FLOAT qualifier on the SQL module processor command line. For IEEE_FLOAT it is DSC$K_DTYPE_FS and for G_FLOAT or D_FLOAT it is DSC$K_DTYPE_F.

[7]The floating point representation of double-precision numbers is determine the setting of the /FLOAT qualifier on the SQL module processor command line. For IEEE_FLOAT it is DSC$K_DTYPE_FS; for G_FLOAT, DSC$K_DTYPE_G; and for D_FLOAT, DSC$K_DTYPE_D.

[9]Not supported by Ada; SQL generates a warning.

(continued on next page)

Table 3–2 (Cont.) SQL and Corresponding OpenVMS Data Types for Module Language

Formal Parameter Data Type	Requires Actual Parameter of OpenVMS Data Type
TIMESTAMP	No equivalent OpenVMS data type; two-longword array
INTERVAL (Year-month)	No equivalent OpenVMS data type; two-longword array
INTERVAL (Day-time)	No equivalent OpenVMS data type; two-longword array
LIST OF BYTE VARYING	Not supported[8]

[8]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string or a quadword. To retrieve the values of individual elements of the list, use host language variables of data type CHAR or VARCHAR.

The following tables show samples for each SQL formal parameter data type and the specific host language declaration SQL accepts for the corresponding actual parameter.

Table 3–3 shows the Ada declarations for SQL formal parameters. Refer to the Usage Note at the end of this section for information about the Ada packages available for the SQL module language. The SQL_STANDARD Ada package defines the data types that are supported by the ANSI/ISO SQL standard.

Table 3–3 Ada Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible Ada Parameter Declaration[2]
CHAR (10)	STR1 : string(1..10);
CHAR (10) CHARACTER SET KANJI	STR1 : string(1..20);[1]
NCHAR (10)	STR1 : string(1..10);[1]
VARCHAR (80)	type VARCHAR_80 is record VAR_LEN : short_integer; VAR_TEXT : array (1..80) of character; end record; STR2 : varchar_80;
VARCHAR (80) CHARACTER SET KANJI	type VARCHAR_160 is record VAR_LEN : short_integer; VAR_TEXT : array (1..160) of character; end record; STR2 : varchar_160;[1]
NCHAR VARYING (80)	type VARCHAR_80 is record VAR_LEN : short_integer; VAR_TEXT : array (1..80) of character; end record; STR2 : varchar_80;[1]
LONG VARCHAR	type VARCHAR_16383 is record VAR_LEN : short_integer; VAR_TEXT : array (1..16383) of character; end record; STR3 : varchar_16383;

[1]See Section 3.3 for information about character length and module language.

[2]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–3 (Cont.) Ada Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible Ada Parameter Declaration[2]
TINYINT (2)	Not supported[3]
TINYINT	NUM1 : short_short_integer;[4]
SMALLINT	Not supported[3]
SMALLINT	NUM1 : short_integer;
INTEGER (2)	Not supported[3]
INTEGER	NUM2 : integer;
BIGINT (2)	Not supported[3]
BIGINT	Not supported[3]
DECIMAL (2)	Not supported[3]
DECIMAL	Not supported[3]
NUMERIC (2)	Not supported[3]
NUMERIC	Not supported[3]
FLOAT (6)	See Table 3–4.
FLOAT (25)	See Table 3–4.
REAL	See Table 3–4.
DOUBLE PRECISION	See Table 3–4.
DATE	Depends on the interpretation of DATE[5]
DATE ANSI	No OpenVMS equivalent[6]
DATE VMS	type SQL_DATE_VMS is record 10 : integer; 11 : integer; end record;
TIME	No OpenVMS equivalent[6]

[2]Assume the default and national character sets of the session are DEC_MCS.

[3]Ada does not support quadword, scaled integer, decimal, or numeric data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in Ada and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[4]This is a DEC Ada extension.

[5]SQL interprets the unqualified DATE data type as DATE VMS by default unless you change the definition environment by specifying DEFAULT DATE FORMAT SQL92 in the SQL module file.

[6]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

(continued on next page)

Table 3–3 (Cont.) Ada Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible Ada Parameter Declaration[2]
TIMESTAMP	No OpenVMS equivalent[6]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[6,7]
LIST OF BYTE VARYING	Not an OpenVMS supported formal parameter data type[8]

[2]Assume the default and national character sets of the session are DEC_MCS.

[6]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[7]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[8]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR or VARCHAR.

Table 3–4 shows the Ada declarations and floating point formats.

Table 3–4 Ada Declarations and Floating Point Formats

Ada Declaration	Compatible SQL$MOD Declaration
pragma FLOAT_REPRESENTATION VAX_FLOAT ... FLOAT1 : STANDARD.FLOAT;	SQLMOD/ADA/FLOAT=G_FLOAT - or SQLMOD/ADA/FLOAT=D_FLOAT ... REAL :P_FLOAT1 - or - FLOAT(24) :P_FLOAT1
pragma FLOAT_REPRESENTATION VAX_FLOAT pragma LONG_FLOAT G_FLOAT ... FLOAT1 : STANDARD.LONG_FLOAT;	SQLMOD/ADA/FLOAT=G_FLOAT ... DOUBLE_PRECISION :P_FLOAT1 - or - FLOAT(53) :P_FLOAT1
pragma FLOAT_REPRESENTATION VAX_FLOAT pragma LONG_FLOAT D_FLOAT ... FLOAT2 : STANDARD.LONG_FLOAT;	SQLMOD/ADA/FLOAT=D_FLOAT ... DOUBLE_PRECISION :P_FLOAT2 - or - FLOAT(53) :P_FLOAT2
... FLOAT1 : SYSTEM.F_FLOAT; FLOAT2 : SYSTEM.D_FLOAT;	SQLMOD/ADA/FLOAT=D_FLOAT ... REAL :P_FLOAT1 - or - FLOAT(24) :P_FLOAT1 DOUBLE_PRECISION :P_FLOAT2 - or - FLOAT(53) :P_FLOAT2

(continued on next page)

Table 3–4 (Cont.) Ada Declarations and Floating Point Formats

Ada Declaration	Compatible SQL$MOD Declaration
... FLOAT1 : SYSTEM.F_FLOAT; FLOAT2 : SYSTEM.G_FLOAT;	SQLMOD/ADA/FLOAT=G_FLOAT ... REAL :P_FLOAT1 - or - FLOAT(24) :P_FLOAT1 DOUBLE_PRECISION :P_FLOAT2 - or - FLOAT(53) :P_FLOAT2
... FLOAT1 : SYSTEM.IEEE_SINGLE_FLOAT; FLOAT2 : SYSTEM.IEEE_DOUBLE_FLOAT;	SQLMOD/ADA/FLOAT=IEEE_FLOAT ... REAL :P_FLOAT1 - or - FLOAT(24) :P_FLOAT1 DOUBLE_PRECISION :P_FLOAT2 - or - FLOAT(53) :P_FLOAT2

Table 3–5 shows the BASIC declarations for SQL formal parameters.

Table 3–5 BASIC Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible BASIC Parameter Declaration[11]
CHAR (10)	DECLARE STRING STR1[1]
CHAR (10) CHARACTER SET KANJI	DECLARE STRING STR1[1,10]
NCHAR (10)	DECLARE STRING STR1[1,10]
VARCHAR (80)	Not supported[2]
VARCHAR (80) CHARACTER SET KANJI	Not supported[2]
NCHAR VARYING (80)	Not supported[2]
LONG VARCHAR	Not supported[2]
TINYINT (2)	Not supported[2]
TINYINT	DECLARE BYTE
SMALLINT (2)	Not supported[2]
SMALLINT	DECLARE WORD NUM1
INTEGER (2)	Not supported[2]
INTEGER	DECLARE LONG NUM2
BIGINT (2)	Not supported[2]
BIGINT	DECLARE QUAD NUM3
DECIMAL(9)	DECLARE DECIMAL (9, 0) NUM7
DECIMAL(18)	DECLARE DECIMAL (18, 0) NUM7
DECIMAL(18,2)	DECLARE DECIMAL (18, 2) NUM7
NUMERIC(2)	Not supported[2]
NUMERIC	Not supported[2]

[1]The BASIC dynamic string data type does not accept an argument for the length of the character string. BASIC passes STRING data by descriptor. For STRING data, SQL ignores the length argument of CHAR formal parameters and uses the descriptor to read (for input) or set (for output) the length of the string.

[2]BASIC does not support varying character, scaled integer, or numeric data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in BASIC and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[10]See Section 3.3 for information about character length and module language.

[11]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–5 (Cont.) BASIC Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible BASIC Parameter Declaration[11]
FLOAT (6)	DECLARE SINGLE NUM4 [3] DECLARE SFLOAT NUM4 [4]
FLOAT (25)	DECLARE DOUBLE NUM4 [3] DECLARE TFLOAT NUM4 [4] DECLARE DFLOAT NUM4 [5]
REAL	DECLARE SINGLE NUM4[3] DECLARE SFLOAT NUM4 [4]
DOUBLE PRECISION	DECLARE DOUBLE NUM4[3] DECLARE TFLOAT NUM4 [4] DECLARE DFLOAT NUM4 [5]
DATE	Depends on the interpretation of DATE[4] DECLARE DATE_REC START_DATE, END_ DATE
DATE VMS	RECORD DATE_REC STRING DATE_STRING=8 END RECORD DATE_REC
DATE ANSI	No OpenVMS equivalent[7] DECLARE DATE_REC START_DATE, END_ DATE
TIME	No OpenVMS equivalent[7] DECLARE TIME_REC START_TIME, END_ TIME
TIMESTAMP	No OpenVMS equivalent[7] DECLARE TIMESTAMP_REC START_ TIMESTAMP, END_TIMESTAMP

[3]If /FLOAT=G_FLOAT is specified on the SQL module processor command line.

[4]If /FLOAT=IEEE_FLOAT is specified on the SQL module processor command line.

[5]If /FLOAT=D_FLOAT is specified on the SQL module processor command line.

[7]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[11]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–5 (Cont.) BASIC Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible BASIC Parameter Declaration[11]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[7,8] DECLARE INTERVAL_REC START_INTERVAL, END_INTERVAL[7]
LIST OF BYTE VARYING	Not supported[9]

[7]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[8]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[9]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR.

[11]Assume the default and national character sets of the session are DEC_MCS.

Table 3–6 shows the C declarations for SQL formal parameters.

Table 3–6 C Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible C Parameter Declaration[10]
CHAR (10)	char str1[11][1,11]
CHAR (10) CHARACTER SET KANJI	char str1[21] [1,8,11]
NCHAR (10)	char str1[11] [1,8,11]

[1]SQL expects character strings to be in ASCIZ format. You therefore declare a CHAR host language variable for a CHAR column to be 1 character more than the column size. (This allows space for the null character that terminates ASCIZ strings.) You can avoid this restriction when you copy definitions from the data dictionary by specifying a character interpretation option in the record-type clause of your parameter declaration.

[8]See Section 3.3 for information about character length and module language.

[10]Assume the default and national character sets of the session are DEC_MCS.

[11]When SQL converts data from a table column to a formal parameter, it fills any extra space in the parameter with blanks. It inserts the null character after the last character or blank-filled space passed from the column to terminate the ASCIZ string.

(continued on next page)

Table 3–6 (Cont.) C Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible C Parameter Declaration[10]
VARCHAR (80)	Dialect dependent[9,11]
VARCHAR (80) CHARACTER SET KANJI	Dialect dependent[9,11]
NCHAR VARYING (80)	Dialect dependent[2,8,9,11]
LONG VARCHAR	Dialect dependent [2,8,11]
TINYINT (2) TINYINT	Not supported[2] char x
SMALLINT (2) SMALLINT	Not supported[2] short num1
INTEGER (2) INTEGER	Not supported[2] int num2
BIGINT (2) BIGINT	not supported[2] _ _int64(2)
DECIMAL (2) DECIMAL	Not supported[2] Not supported[2]
NUMERIC (2) NUMERIC	Not supported[2] Not supported[2]
FLOAT (6) FLOAT (25)	float num4[3] double num4[3]

[2]C does not support varying character, scaled integer, decimal, or numeric data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in C and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[3]The/FLOAT or /[NO]G_FLOAT qualifier on SQL module processor command line must match that for C compiling.

[8]See Section 3.3 for information about character length and module language.

[9]Although C does not support varying character data types, if you specify DIALECT SQL92 and C language, you can declare formal parameters as VARCHAR, NCHAR VARYING, and LONG VARCHAR. SQL passes the parameters as ASCIZ (null-terminated string). If you specify DIALECT SQLV40 or SQL89, SQL passes the parameters as ASCIW (word length prefixed) and returns a deprecated feature message. If you specify DIALECT MIA, SQL passes the parameters as ASCIW but does not issue a deprecated feature message because MIA dictates that these parameters are passed this way. If you do not specify a dialect, SQL passes the parameters as ASCIW (word length prefixed) and returns a deprecated feature message.

[10]Assume the default and national character sets of the session are DEC_MCS.

[11]When SQL converts data from a table column to a formal parameter, it fills any extra space in the parameter with blanks. It inserts the null character after the last character or blank-filled space passed from the column to terminate the ASCIZ string.

(continued on next page)

Table 3–6 (Cont.) C Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible C Parameter Declaration[10]
REAL	float num5[3]
DOUBLE PRECISION	double num6[3]
DATE	Depends on the interpretation of DATE[4]
DATE ANSI	No OpenVMS equivalent[5]
DATE VMS	struct { int 10; int 11; }sql_date_vms;
TIME	No OpenVMS equivalent[5]
TIMESTAMP	No OpenVMS equivalent[5]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[5,6]
LIST OF BYTE VARYING	Not supported[7]

[3]The/FLOAT or /[NO]G_FLOAT qualifier on SQL module processor command line must match that for C compiling.

[4]SQL interprets the unqualified DATE data type as DATE VMS by default unless you change the definition environment by specifying DEFAULT DATE FORMAT SQL92 in the SQL module file.

[5]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[6]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[7]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR.

[10]Assume the default and national character sets of the session are DEC_MCS.

Table 3–7 shows the COBOL declarations for SQL formal parameters.

Table 3–7 COBOL Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible COBOL Parameter Declaration[7]
CHAR (10)	01 STR1 PICTURE X(10).
CHAR (10) CHARACTER SET KANJI	01 STR1 PICTURE X(20).[6]
NCHAR (10)	01 STR1 PICTURE X(10).[6]

VARCHAR (80)	`01 STR2.`
	`  49 STR2L PICTURE S9(4) COMP.`
	`  49 STR2C PICTURE X(80).`

VARCHAR (80) CHARACTER SET KANJI	`01 STR2 CHARACTER SET KANJI.`
	`  49 STR2L PICTURE S9(4) COMP.`
	`  49 STR2C PICTURE X(160).`[6]

NCHAR VARYING (80)	`01 STR2.`
	` 49 STR2L PICTURE S9(4) COMP.`
	` 49 STR2C PICTURE X(80).`[6]

LONG VARCHAR	`01 STR3.`
	`  49 STR3L PICTURE S9(4) COMP.`
	`  49 STR3C PICTURE X(16383).`

SMALLINT (2)	01 NUM1 PICTURE S99V99 COMP.
SMALLINT	01 NUM1 PICTURE S9(4) COMP.
INTEGER (2)	01 NUM2 PICTURE S9(7)V99 COMP.
INTEGER	01 NUM2 PICTURE S9(9) COMP.
BIGINT (2)	01 NUM3 PIC S9(16)V99 COMP.
BIGINT	01 NUM3 PIC S9(18) COMP.
DECIMAL(18,2)	01 NUM4 PIC S9(16)V99 COMP3.
DECIMAL(18)	01 NUM4 PIC S9(18) COMP3.

[6]See Section 3.3 for information about character length and module language.

[7]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–7 (Cont.) COBOL Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible COBOL Parameter Declaration[7]
NUMERIC(18,2) NUMERIC(18)	01 NUM5 PIC S9(16)V99 SIGN LEADING SEPARATE. 01 NUM5 PIC S9(18) SIGN LEADING SEPARATE.
FLOAT (6) FLOAT (25)	01 NUM6 COMP-1.[1] 01 NUM6 COMP-2.[1]
REAL	01 NUM7 COMP-1. [1]
DOUBLE PRECISION	01 NUM8 COMP-2.[1]
DATE	Depends on the interpretation of DATE[2]
DATE ANSI	No OpenVMS equivalent[3]

DATE VMS	`type SQL_DATE_VMS is`
	`  record`
	`  10 : integer;`
	`  11 : integer;`
	`  end record;`

TIME	No OpenVMS equivalent[3]
TIMESTAMP	No OpenVMS equivalent[3]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[3,4]
LIST OF BYTE VARYING	Not supported[5]

[1]The /FLOAT or /[NO]G_FLOAT qualifier on SQL module processor command line must match that for COBOL compiling.

[2]SQL interprets the unqualified DATE data type as DATE VMS by default unless you change the definition environment by specifying DEFAULT DATE FORMAT SQL92 in the SQL module file.

[3]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[4]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[5]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string identifier, a pointer to the first element of the list, using a quadword or an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR or VARCHAR.

[7]Assume the default and national character sets of the session are DEC_MCS.

Table 3–8 shows the FORTRAN declarations for SQL formal parameters.

Table 3–8 FORTRAN Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible FORTRAN Parameter Declaration[9]
CHAR (10)	CHARACTER*10 STR1
CHAR (10) CHARACTER SET KANJI	CHARACTER*20 STR1[8]
NCHAR (10)	CHARACTER*10 STR1[8]
VARCHAR (80)	Not supported[1]
VARCHAR (80) CHARACTER SET KANJI	Not supported[1]
NCHAR VARYING (80)	Not supported[1]
LONG VARCHAR	Not supported[1]
TINYINT (2)	Not supported[1]
TINYINT	LOGICAL*1[2]
SMALLINT (2)	Not supported[1]
SMALLINT	INTEGER*2 NUM1
INTEGER (2)	Not supported[1]
INTEGER	INTEGER*4 NUM2
BIGINT (2)	Not supported[1]
BIGINT	Not supported[1]
DECIMAL(18,2)	Not supported[1]
DECIMAL(18)	Not supported[1]
NUMERIC(18,2)	Not supported[1]
NUMERIC(18)	Not supported[1]

[1]FORTRAN does not support varying character, scaled integer, or numeric data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in FORTRAN and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[2]In FORTRAN, BYTE is a synonym for LOGICAL*1 and is parsed by the SQL interface for Oracle Rdb.

[8]See Section 3.3 for information about character length and module language.

[9]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–8 (Cont.) FORTRAN Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible FORTRAN Parameter Declaration[9]
FLOAT (6) FLOAT (25)	REAL*4 NUM4[3] DOUBLE PRECISION NUM4[3]
REAL	REAL*4 NUM5[3]
DOUBLE PRECISION	DOUBLE PRECISION NUM6[3]
DATE	Depends on the interpretation of DATE[4]
DATE ANSI	No OpenVMS equivalent[5]
DATE VMS	STRUCTURE /SQL_DATE_VMS/ INTEGER*4 L0 INTEGER*4 L1 END STRUCTURE
TIME	No OpenVMS equivalent[5]
TIMESTAMP	No OpenVMS equivalent[5]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[5,6]
LIST OF BYTE VARYING	Not supported[7]

[3]The /FLOAT or /[NO]G_FLOAT qualifier on SQL module processor command line must match that for FORTRAN compiling.

[4]SQL interprets the unqualified DATE data type as DATE VMS by default unless you change the definition environment by specifying DEFAULT DATE FORMAT SQL92 in the SQL module file.

[5]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[6]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[7]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string (list) identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR.

[9]Assume the default and national character sets of the session are DEC_MCS.

Table 3–9 shows the Pascal declarations for SQL formal parameters.

Table 3–9 Pascal Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible Pascal Parameter Declaration[11]
CHAR (10)	VAR STR1 : PACKED ARRAY [1..10] OF CHAR;
CHAR (10) CHARACTER SET KANJI	VAR STR1 : PACKED ARRAY [1..20] OF CHAR;[10]
NCHAR (10)	VAR STR1 : PACKED ARRAY [1..10] OF CHAR;[10]
VARCHAR (80)	VAR STR2 : VARYING [80] OF CHAR;
VARCHAR (80) CHARACTER SET KANJI	VAR STR2 : VARYING [160] OF CHAR;[10]
NCHAR VARYING (80)	VAR STR2 : VARYING [80] OF CHAR;[10]
LONG VARCHAR	VAR STR3 : VARYING [16383] OF CHAR;
TINYINT (2) TINYINT	Not supported[1] VAR NUM1 : [BYTE] –128..127;
SMALLINT (2) SMALLINT	Not supported[1] VAR NUM1 : [WORD] –32768..32767;
INTEGER (2) INTEGER	Not supported[1] VAR NUM2 : [LONG] –MAXINT..+MAXINT;
BIGINT (2) BIGINT	Not supported[1] Not supported[1]
DECIMAL(18,2) DECIMAL(18)	Not supported[1] Not supported[1]
NUMERIC(18,2) NUMERIC(18)	Not supported[1] Not supported[1]

[1]Pascal does not support packed decimal, numeric, or scaled integer data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in Pascal and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[10]See Section 3.3 for information about character length and module language.

[11]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–9 (Cont.) Pascal Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible Pascal Parameter Declaration[11]
FLOAT (6)	VAR NUM4 : SINGLE [2] VAR NUM4 : F_FLOAT [3,4] VAR NUM4 : S_FLOAT [5]
FLOAT (25)	VAR NUM4 : DOUBLE [2] VAR NUM4 : G_FLOAT [3] VAR NUM4 : D_FLOAT [4] VAR NUM4 : T_FLOAT [5]
REAL	VAR NUM4 : SINGLE [2] VAR NUM4: F_FLOAT [3,4] VAR NUM4: S_FLOAT [5]
DOUBLE PRECISION	VAR NUM4 : DOUBLE [2] VAR NUM4 : G_FLOAT [3] VAR NUM4 : D_FLOAT [4] VAR NUM4 : T_FLOAT [5]
DATE	Depends on the interpretation of DATE[6]
DATE ANSI	No OpenVMS equivalent[7]
DATE VMS	SQL_DATE_VMS = RECORD L0 : INTEGER; L1 : INTEGER; END;
TIME	No OpenVMS equivalent[7]
TIMESTAMP	No OpenVMS equivalent[7]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[7,8]

[2]If The /FLOAT or /[NO]G_FLOAT qualifier on SQL module processor command line must match that for Pascal compiling.

[3]If /FLOAT=G_FLOAT is specified on the SQL module processor command line.

[4]If /FLOAT=D_FLOAT is specified on the SQL module processor command line.

[5]If /FLOAT=IEEE_FLOAT is specified on the SQL module processor command line.

[6]SQL interprets the unqualified DATE data type as DATE VMS by default unless you change the definition environment by specifying DEFAULT DATE FORMAT SQL92 in the SQL module file.

[7]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[8]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[11]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–9 (Cont.) Pascal Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible Pascal Parameter Declaration[11]
LIST OF BYTE VARYING	Not supported[9]

[9]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR or VARCHAR.

[11]Assume the default and national character sets of the session are DEC_MCS.

Table 3–10 shows the PL/I declarations for SQL formal parameters.

Table 3–10 PL/I Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible PL/I Parameter Declaration[9]
CHAR (10)	DCL STR1 CHAR(10);
CHAR (10) CHARACTER SET KANJI	DCL STR1 CHAR(20);[8]
NCHAR (10)	DCL STR1 CHAR(10);[8]
VARCHAR (80)	DCL STR2 CHAR(80) VAR;
VARCHAR (80) CHARACTER SET KANJI	DCL STR2 CHAR(160) VAR;[8]
NCHAR VARYING (80)	DCL STR2 CHAR(80) VAR;[8]
LONG VARCHAR	DCL STR3 CHAR(16383) VAR;
TINYINT (2) TINYINT	Not supported[1] FIXED BINARY(7);
SMALLINT (2) SMALLINT	Not supported[1] DCL NUM1 BIN FIXED(15);
INTEGER (2) INTEGER	Not supported[1] DCL NUM2 BIN FIXED(31);

[1]PL/I does not support scaled integer or BIGINT data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in PL/I and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[8]See Section 3.3 for information about character length and module language.

[9]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–10 (Cont.) PL/I Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible PL/I Parameter Declaration[9]
BIGINT (2)	Not supported[1]
BIGINT	Not supported[1]
DECIMAL(4)	DCL NUM3 FIXED(4) DEC;
DECIMAL(18,2)	DCL NUM3 FIXED(18,2) DEC;
NUMERIC(2)	DCL NUM4 PIC 'S(4)9';[2]
NUMERIC(18,2)	DCL NUM4 PIC 'S(16)9V99';[2]
FLOAT (6)	DCL NUM5 BIN FLOAT(24);[3]
FLOAT (25)	DCL NUM5 BIN FLOAT(53);[3]
REAL	DCL NUM5 BIN FLOAT(24);[3]
DOUBLE PRECISION	DCL NUM5 BIN FLOAT(53);[3]
DATE	Depends on the interpretation of DATE[4]
DATE ANSI	No OpenVMS equivalent[5]
DATE VMS	DECLARE 1 SQL_DATE_VMS, 2 L0 BIN FIXED(31); 2 L1 BIN FIXED(31);
TIME	No OpenVMS equivalent[5]
TIMESTAMP	No OpenVMS equivalent[5]
INTERVAL DAY TO SECOND	No OpenVMS equivalent[5,6]

[1]PL/I does not support scaled integer or BIGINT data types. To retrieve columns defined with those data types from a database, declare formal parameters with a data type that is supported in PL/I and refer to those formal parameters in SQL module procedure statements. SQL will convert the data in the columns to the data type of the formal parameter.

[2]Do not pass the data type by descriptor.

[3]The /FLOAT or /[NO]G_FLOAT qualifier on SQL module processor command line must match that for PL/I compiling.

[4]SQL interprets the unqualified DATE data type as a DATE VMS by default unless you change the definition environment by specifying DEFAULT DATE FORMAT SQL92 in the SQL module file.

[5]Except for DATE VMS, the length and format of the date-time data types are reserved for use by Oracle Rdb. Use the data types shown in Table 4–2 in host programs.

[6]The INTERVAL data type has 12 other qualifier combinations listed in Table 2–9.

[9]Assume the default and national character sets of the session are DEC_MCS.

(continued on next page)

Table 3–10 (Cont.) PL/I Declarations for SQL Formal Parameter Data Types

SQL Formal Parameter Data Type	Compatible PL/I Parameter Declaration[9]
LIST OF BYTE VARYING	Not supported[7]

[7]Module language does not support LIST OF BYTE VARYING as a formal parameter data type. However, you can retrieve the segmented string (list) identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of the list, use host language variables of data type CHAR or VARCHAR.

[9]Assume the default and national character sets of the session are DEC_MCS.

Usage Note

The SQL module language provides support for three Ada packages:

- SQL_STANDARD
- SQL_SQLCODE
- SQL_SQLCA

The SQL_SQLCA package defines the SQLCA structure, and the SQL_SQLCODE package contains the literal definitions for the SQLCODE values.

Note

Ada literals can contain only ASCII characters.

SQL lets you declare host language variables either directly or by calling the SQL_STANDARD Ada package.

You must use the SQL_STANDARD Ada package if you want to conform to the ANSI/ISO SQL standard. This package defines the data types that are supported by the ANSI standard. To use the package, first copy the file SYS$LIBRARY:SQL$STANDARD.ADA to your own Ada library, then compile the package.

The package SQL_STANDARD declares the following ANSI-standard data types:

- CHAR
- SMALLINT
- INT

- REAL
- DOUBLE_PRECISION
- SQLCODE_TYPE

The data type SQLCODE_TYPE contains three subtypes: NOT_FOUND, INDICATOR_TYPE, and SQL_ERROR.

When you compile an SQL module using Ada as the source language, Oracle Rdb generates an Ada package that contains Ada declarations for all procedures in the SQL module. Part of that declaration for each routine is the declaration of each parameter. These parameters will be declared using data types in the SQL_STANDARD package.

You must use the Ada WITH clause in your host language program to take advantage of this generated package. This generated package has the same name as your SQL module.

To take advantage of this generated package, be sure to compile your SQL module before compiling your host language program. When using ACS LINK, first specify the Ada source program object file, then the object file created by the SQL module, and then any other libraries you might need, such as SQL$USER.

Examples

Example 1: Compiling an SQL module file using the generated Ada package

```
MODULE MY_MODULE 1
LANGUAGE ADA
AUTHORIZATION RDB$DBHANDLE

DECLARE ONE SCHEMA FILENAME personnel_one
DECLARE TWO SCHEMA FILENAME personnel_two

   .
   .
   .
PROCEDURE COUNT
    SQLCODE
    THE_COUNT INT;

    SELECT COUNT(DISTINCT EMPLOYEE_ID)
        INTO THE_COUNT
        FROM ONE.EMPLOYEES;
```

1 Note the SQL module name.

Example 2: Showing the object file generated by the SQL module language compiler in Example 1

```
   .
   .
   .
--Source file is USER1:[ADA]MY_MODULE.SQLMOD;1
WITH SQL_STANDARD;  1
WITH SYSTEM;
Package MY_MODULE is  2

PROCEDURE COUNT (
                P1 : in out SQL_STANDARD.SQLCODE_TYPE; 3
                P2 : in out SQL_STANDARD.INT   4
        );

pragma INTERFACE (NONADA, COUNT);

End MY_MODULE;
```

1 The SQL_STANDARD Ada package is being called.

2 The SQL module name is specified.

3 Using SQLCODE_TYPE data type from SQL_STANDARD.

4 Using INT data type from SQL_STANDARD package.

3.6 SQL Module Language Processor Command Line

You can define a symbol to make invoking the SQL module processor easier. For example:

```
$ SQLMOD == "$SQL$MOD"
```

You then can invoke the SQL module processor with or without a module file specification:

- If you invoke the SQL module processor without a module file specification, the module processor prompts you for it. For example:

  ```
  $ SQLMOD
  INPUT FILE> module-file-specification
  ```

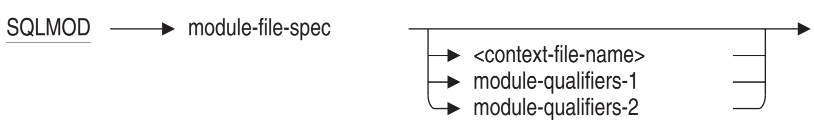
- If you invoke the SQL module processor with a module file specification as part of the DCL command line, SQL starts processing your module file immediately after you press the Return key. For example:

  ```
  $ SQLMOD module-file-specification
  ```

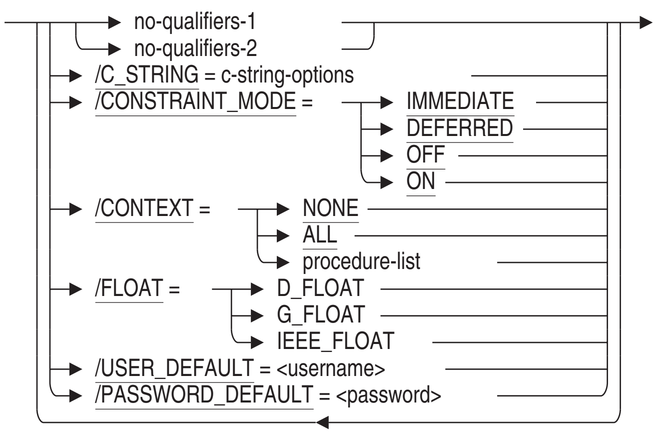
Either way, there are several qualifiers you can specify with the file specification that control how SQL processes the module file. The syntax diagram shows the format for those qualifiers.

Format

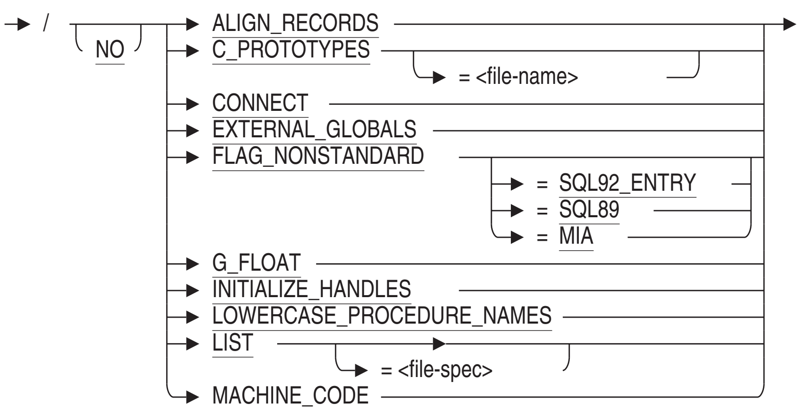
module-file-spec-qual =



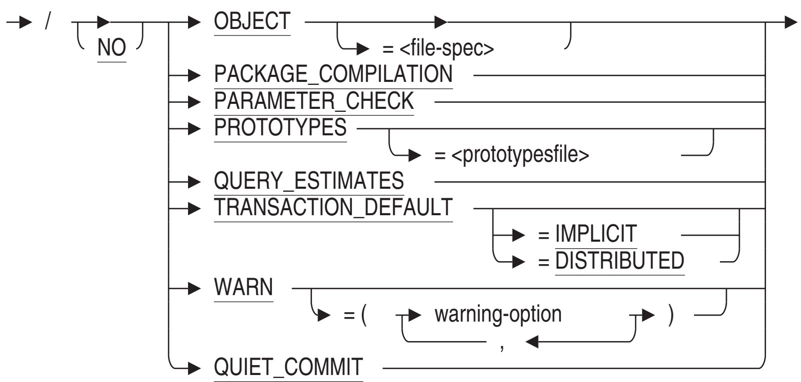
module-qualifiers-1 =



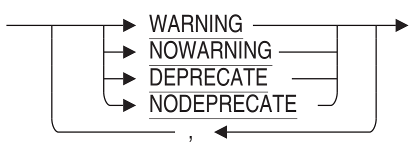
no-qualifiers-1 =



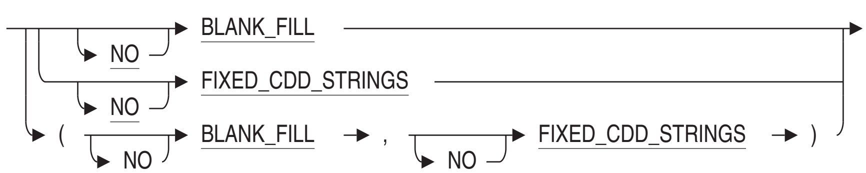
no-qualifiers-2 =



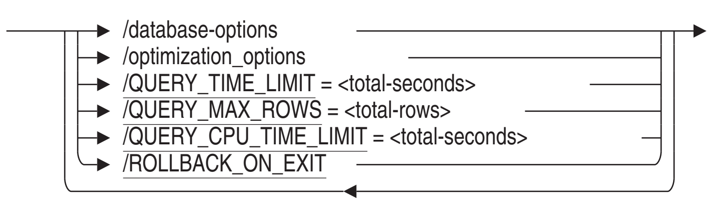
warning-option =



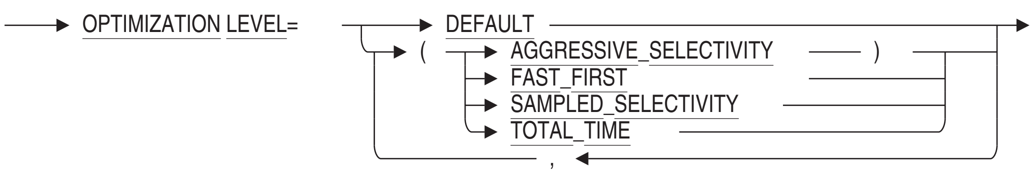
c-string-options =



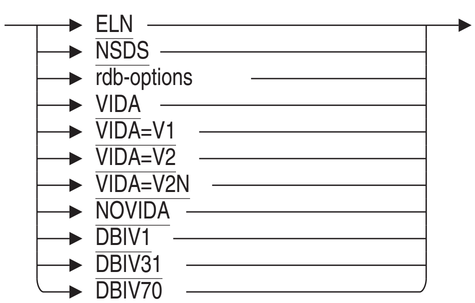
module-qualifiers-2 =



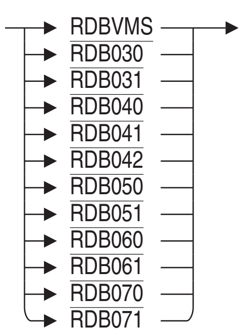
optimization-options=



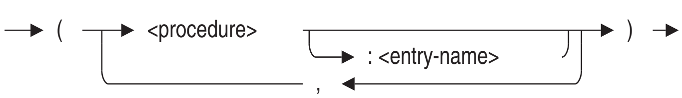
database-options =



rdb-options =



procedure-list =



Arguments

context-file-name
module-file-spec
The file specification for an SQL module source file. The default file extension for the source file is .sqlmod.

The context-file-name is an SQL command procedure containing DECLARE statements that you want to apply when your program compiles and executes. See Section 2.11 for information about context-file-name.

module-qualifiers-1
A set of qualifiers that you can optionally apply to the SQL module processor command line.

no-qualifiers-1
You can add the NO prefix to negate any qualifier in this group.

ALIGN_RECORDS
NOALIGN_RECORDS
Aligns the fields in an SQL module procedure record parameter.

If you are using the OpenVMS Alpha platform and if your host language is C, the default is ALIGN_RECORDS; otherwise, the default on the OpenVMS Alpha platform is NOALIGN_RECORDS.

On the OpenVMS VAX platform, the default is NOALIGN_RECORDS.

C_PROTOTYPES=file-name
NOC_PROTOTYPES
This qualifier is deprecated and will not be supported in a future release. It has been replaced by the PROTOTYPES qualifier.

CONNECT
NOCONNECT
Specifies whether or not SQL allows multiple user sessions and access to global databases across modules. All SQL modules in an application must be compiled with connections enabled or disabled.

The default setting is NOCONNECT.

EXTERNAL_GLOBALS
NOEXTERNAL_GLOBALS
Specifies whether or not alias references are coerced into alias definitions. An **alias definition** is an alias declared with the GLOBAL keyword (the default)

in the DECLARE ALIAS statement. An **alias reference** is an alias declared with the EXTERNAL keyword in the DECLARE ALIAS statement.

The EXTERNAL_GLOBALS qualifier treats alias references as alias definitions. This qualifier provides compatibility with versions prior to V7.0.

The NOEXTERNAL_GLOBALS qualifier treats alias references as alias references. The NOEXTERNAL_GLOBALS qualifier may be useful on OpenVMS if your application shares an alias between multiple shareable images.

The default setting is EXTERNAL_GLOBALS.

See the DECLARE ALIAS Statement for more information about alias definitions and references. For information on using aliases and shareable images, see the *Oracle Rdb Guide to SQL Programming*.

FLAG_NONSTANDARD
FLAG_NONSTANDARD=SQL92_ENTRY
FLAG_NONSTANDARD=SQL89
FLAG_NONSTANDARD=MIA
NOFLAG_NONSTANDARD

Specifies whether or not SQL identifies nonstandard syntax. Nonstandard syntax, called an extension, refers to syntax that is not part of the ANSI/ISO SQL standard or the Multivendor Integration Architecture (MIA) standard. You can specify the following options:

- FLAG_NONSTANDARD

  Notifies you of syntax that is an extension to the ANSI/ISO SQL standard.

- FLAG_NONSTANDARD=SQL92_ENTRY

  Notifies you of syntax that is an extension to the ANSI/ISO SQL standard. This qualifier has the same effect on flagging as does the FLAG_NONSTANDARD qualifier.

- FLAG_NONSTANDARD=SQL89

  Notifies you of syntax that is an extension to the ANSI/ISO 1989 standard.

- FLAG_NONSTANDARD=MIA

  Notifies you of syntax that is an extension to the MIA standard.

- NOFLAG_NONSTANDARD

  Prevents notification of extensions.

Preventing notification of extensions (NOFLAG_NONSTANDARD) is the default.

FLOAT=D_FLOAT
FLOAT=G_FLOAT
FLOAT=IEEE_FLOAT

The /FLOAT qualifier determines the conversion that SQL Module language performs on SQL Module Language procedure parameters declared as single or double precision floating point SQL datatypes. SQL floating point datatypes are FLOAT(n), REAL, and DOUBLE PRECISION. See Section 2.3 for details. Internally to Oracle Rdb, single precision floating point types are represented as F-Floating while double precision floating point types are represented and G-Floating. See Table 3–2 for more details.

By default, parameters declared as single or double precision floating point type are expected to be passed by the calling host language program in F-Floating and G-Floating format, respectively. This is equivalent to using a qualifier of /FLOAT=G_FLOAT with the SQL$MOD command.

If the command line for SQL$MOD has /FLOAT=D_FLOAT, then the single and double precision floating point parameters are expected to be in F-Floating and D-Floating format respectively. SQL Module Language will convert the double precision parameters between D-Floating and G-Floating formats for both input and output.

If the command line for SQL$MOD has /FLOAT=IEEE_FLOAT, the single and double precision floating point parameters are expected to be in IEEE S-Floating and IEEE T-Floating format, respectively. SQL Module Language will convert between these formats and the internal F-Floating and G-Floating formats for both input and output.

If a parameter of an SQL Module Language procedure is of a record type, any fields of the record which are of floating point types follow the same rules as described above.

The floating point formats of the host language program actual parameters must agree with the format expected by the SQL Module Language actual parameter. (See Section 3.5 for information concerning actual and formal parameter agreement.)

Notes

Oracle Rdb always stores floating point numbers internally using the VAX 32-bit and 64-bit types called F-Floating (F_FLOAT) and G-Floating (G_FLOAT), respectively. This means that when IEEE formats are used in a host language program, Oracle Rdb converts back and forth between the VAX and IEEE formats. There are differences in the number of available bits in the fraction and exponent between these formats. Additionally, the IEEE formats have certain exponent values reserved for infinity values. These differences can cause floating point overflow or underflow as well as rounding errors during the conversion process. See Appendix A of the Portable Mathematics Library in the OpenVMS Operating System documentation for data on the maximum and minimum values for VAX versus IEEE floating point formats.

When /FLOAT=IEEE_FLOAT is used, floating point data types may not be imported from the Common Data Dictionary.

G_FLOAT
NOG_FLOAT

The /G_FLOAT and /NOG_FLOAT qualifiers are for backwards compatibility. They are equivalent to /FLOAT=G_FLOAT and /FLOAT=D_FLOAT, respectively. You should not specify both /FLOAT and /[NO]G_FLOAT qualifiers.

INITIALIZE_HANDLES
NOINITIALIZE_HANDLES

Specifies whether or not alias definitions are coerced into alias references. The NOINITIALIZE_HANDLES qualifier causes all alias declarations to be treated as alias references.

An **alias definition** is an alias declared with the GLOBAL keyword (the default) in the DECLARE ALIAS statement. An **alias reference** is an alias declared with the EXTERNAL keyword in the DECLARE ALIAS statement.

The NOINITIALIZE_HANDLES qualifier may be useful for existing source code on OpenVMS in coercing alias definitions into alias references. Because there is usually no distinction between a definition and a reference on OpenVMS, your application might declare an alias definition where an alias reference is needed. If you reorganize your application into multiple images that share aliases, you must distinguish the alias definition from the alias reference. In this case, use the NOINITIALIZE_HANDLES qualifier to coerce a definition into a reference without changing your source code.

If your application correctly declares alias references with the EXTERNAL keyword, use the NOEXTERNAL_GLOBALS qualifier, instead of the [NO]INITIALIZE_HANDLES to override the default on OpenVMS and cause SQL to treat alias references properly as references.

The default setting is INITIALIZE_HANDLES. This qualifier overrides the [NO]EXTERNAL_GLOBALS qualifier.

This qualifier is maintained for compatibility with previous versions of Oracle Rdb. For V7.0 and higher, use the [NO]EXTERNAL_GLOBALS qualifier, which provides more precise control over alias definition. See the DECLARE ALIAS Statement for more information about alias definitions and references. For information on using aliases and shareable images, see the *Oracle Rdb Guide to SQL Programming*.

LOWERCASE_PROCEDURE_NAMES
NOLOWERCASE_PROCEDURE_NAMES

Forces the names of the module language procedures to be in lowercase. This qualifier not only assumes that the SQL module procedure names are in lowercase, it overrides the case in any quoted SQL module procedure.

The default setting is NOLOWERCASE_PROCEDURE_NAMES.

LIST
NOLIST

Determines whether or not the SQL module processor creates a list file containing the original module list along with any error messages from the processing, and, if it does, what it is named. The NOLIST qualifier is the default. If you specify the LIST qualifier and do not include a file specification, the SQL module processor creates a list file with the same file name as your module source file with the file extension .lis.

MACHINE_CODE
NOMACHINE_CODE

With Oracle Rdb for OpenVMS Alpha, determines whether or not the SQL module processor includes machine code in the list (.lis) file; however, to generate the list file with the machine code in it, you must also specify the LIST qualifier.

The NOMACHINE_CODE qualifier is the default.

no-qualifiers-2

You can add the NO prefix to negate any qualifier in this group.

OBJECT
NOOBJECT

Specifies whether or not the SQL module processor creates an object file when compiling the source file if the compilation does not generate fatal errors; and, if an object file is produced, what the file is named. The OBJECT qualifier is the default. If you specify the OBJECT qualifier and do not include a file specification, the SQL module processor creates an object file with the same file name as the source file and with the file extension .obj.

PACKAGE_COMPILATION
NOPACKAGE_COMPILATION
Determines if a package specification is produced and loaded into the ACS library.

Oracle Rdb produces a package specification when you process a module with the LANGUAGE ADA clause specified in the module header unless you specify the NOPACKAGE_COMPILATION qualifier. The NOPACKAGE_COMPILATION qualifier prevents the package specification from being loaded in the ACS library, but still creates and compiles the .ada file.

The PACKAGE_COMPILATION qualifier is the default.

PARAMETER_CHECK
NOPARAMETER_CHECK
Specifies whether or not the SQL module processor compares the number of formal parameters declared for a procedure with the number of parameters specified in the SQL statement of the procedure:

- PARAMETER_CHECK (default)

  Checks that parameter counts match and generates an error at run time (not compile time) when they do not.

- NOPARAMETER_CHECK

  Suspends checking parameters to improve module compilation time. Consider using the NOPARAMETER_CHECK qualifier after you have debugged your SQL module.

SQL checks parameter counts by default. To improve module compilation time, you must explicitly use the NOPARAMETER_CHECK qualifier.

PROTOTYPES[=prototypesfile]
NOPROTOTYPES
The PROTOTYPES qualifier uses the LANGUAGE clause from the module to generate routine declarations for the following languages: C (C++), Pascal, and BLISS. The qualifier is ignored for all other language values.

The prototypes file specification defaults to the same device, directory, and file name as the module language source. The file types default to .h for C, .PAS for Pascal, and .REQ for BLISS.

For the BLISS language, the PROTOTYPES qualifier generates EXTERNAL ROUTINE declarations for each SQL module language procedure.

For the Pascal language, the generated external procedure declarations are suitable for inclusion in either a Pascal program or module. Structured types (RECORD ... END RECORD), SQLDA, and SQLCA used by the SQL module language procedures are declared as UNSAFE arrays of bytes to simplify passing structures via these external definitions. However, care must be taken as this form of declaration disables the strong typing checks in Pascal.

The output for the C language includes pre-processor directives to conditionally include C++ "extern C" syntax and also allow multiple #include references.

The default setting is NOPROTOTYPES.

QUERY_ESTIMATES
NOQUERY_ESTIMATES

Specifies whether or not SQL returns the estimated number of rows and estimated number of disk I/O operations in the SQLCA structure. If you specify the default, which is the QUERY_ESTIMATES qualifier, SQL returns the estimated number of rows in the field SQLCA.SQLERRD[2] and the estimated number of disk I/O operations in the field SQLCA.SQLERRD[3]. The value of SQLCA.SQLERRD[2] and SQLCA.SQLERRD[3] is normally 0 after you execute an OPEN statement for a table.

The following example shows interactive SQL output from a statement that accesses the INTRO_PERSONNEL database. The database was loaded using the sample program SQL$INTRO_LOAD_EMPL_C.SQLMOD with the QUERY_ESTIMATES qualifier on the module language command line. The SQLCA.SQLERRD[2] field shows that SQL estimates 100 rows. The SQLCA.SQLERRD[3] field shows that SQL estimates 16 disk I/O operations.

```
$ SQL$
SQL> ATTACH 'FILENAME intro_personnel';
SQL> DECLARE MY_CURSOR
cont> TABLE CURSOR FOR
cont> SELECT * FROM EMPLOYEES;
SQL> OPEN MY_CURSOR;
SQL> SHOW SQLCA;
SQLCA:
        SQLCAID:        SQLCA           SQLCABC:        128
        SQLCODE:        0
        SQLERRD:        [0]: 0
                        [1]: 0
                        [2]: 100
                        [3]: 16
                        [4]: 0
                        [5]: 0
        SQLWARN0:               SQLWARN1:               SQLWARN2:
        SQLWARN3:               SQLWARN4:               SQLWARN5:
        SQLWARN6:               SQLWARN7:
```

TRANSACTION_DEFAULT=IMPLICIT
TRANSACTION_DEFAULT=DISTRIBUTED
NOTRANSACTION_DEFAULT

Specifies when SQL starts a transaction and how SQL handles default distributed transactions. You can specify the following options:

- TRANSACTION_DEFAULT=IMPLICIT

  Causes SQL to start a transaction when you issue either a SET TRANSACTION statement or the first executable SQL statement in a session.

- TRANSACTION_DEFAULT=DISTRIBUTED

  Causes SQL to use the distributed transaction identifier (TID) for the default distributed transaction established by the DECdtm system service SYS$START_TRANS. Using this option eliminates the need to declare context structures in host language programs and to pass context structures to SQL module procedures. Because it closes all cursors, it also eliminates the need to call the SQL_CLOSE_CURSORS routine.

  You must explicitly call the DECdtm system services when you use this option.

  This option provides support for the **Structured Transaction Definition Language (STDL)** of the Multivendor Integration Architecture (MIA) standard.

  If you specify the TRANSACTION_DEFAULT=DISTRIBUTED option with the CONTEXT qualifier, you must declare a context structure and pass the context structure to the statements named in the CONTEXT qualifier or, if you specify CONTEXT=ALL, to most executable statements involved in the distributed transaction. See Section 2.9 for information about which executable statements do not require a context structure.

- NOTRANSACTION_DEFAULT

  Prevents SQL from starting a transaction unless you execute a SET TRANSACTION statement. If you use this qualifier and issue an executable statement without first issuing a SET TRANSACTION statement, SQL returns an error.

The default is TRANSACTION_DEFAULT=IMPLICIT.

warning-option

Specifies whether the SQL module processor writes informational and warning messages to your terminal, a list file, or both. The WARN qualifier is the default. You can specify two warning options with the WARN qualifier to customize message output.

You cannot specify warning options if you specify the NOWARN qualifier.

WARNING
NOWARNING
You can use combinations of the warning options to specify which warning messages the SQL module processor writes. If you specify only a single warning option, you do not need the parentheses.

The WARNING and NOWARNING qualifiers specify whether or not the SQL module processor writes informational and warning messages.

DEPRECATE
NODEPRECATE
The DEPRECATE and NODEPRECATE qualifiers specify whether or not the SQL module processor writes diagnostic messages about deprecated features.

Deprecated features are currently allowed features that will not be allowed in future versions of SQL; that is, they will be obsolete. For example, one deprecated feature is the use of obsolete keywords such as VERB_TIME instead of VERB TIME. A complete list of deprecated features appears on line in the interactive SQL Help utility.

You can specify the WARN=WARNING qualifier if you prefer to have all warning messages except those about deprecated features. You can specify the WARN=(NOWARNING, DEPRECATE) qualifier if you prefer only the deprecated feature messages. The WARN qualifier alone is equivalent to the WARN=(WARNING, DEPRECATE) qualifier, which means that SQL writes informational and warning messages, plus messages about deprecated features. The NOWARN qualifier alone is equivalent to the WARN=(NOWARNING, NODEPRECATE) qualifier, which means that SQL does not write any messages.

c-string-options
Controls how SQL handles C host language character strings.

Use either or both of the [NO]BLANK_FILL and [NO]FIXED_CDD_STRINGS keywords with the C_STRING qualifier to control C string characteristics.

C_STRING=[NO]BLANK_FILL
C_STRING=[NO]FIXED_CDD_STRINGS
C_STRING=([NO]BLANK_FILL,[NO]FIXED_CDD_STRINGS)
Specifies how to handle C host language character strings:

- [NO]BLANK_FILL (default: BLANK_FILL)

Controls whether or not C character strings are filled with blanks as required by the SQL89 and ANSI/ISO SQL standards or if the null terminator is placed after the last data byte of the source string.

- [NO]FIXED_CDD_STRINGS (default: NOFIXED_CDD_STRINGS)

  Controls whether or not SQL treats C character strings from Oracle CDD/Repository record definitions as fixed-length character strings or C null-terminated strings.

CONSTRAINT_MODE=IMMEDIATE
CONSTRAINT_MODE=DEFERRED
CONSTRAINT_MODE=ON
CONSTRAINT_MODE=OFF

You can optionally specify either the CONSTRAINT_MODE=IMMEDIATE or CONSTRAINT_MODE=DEFERRED qualifier on the SQL module language command line to set the default constraint evaluation mode for commit-time constraints. (This qualifier does not affect the evaluation of verb-time constraints.) The default is DEFERRED; that is, commit-time constraints are evaluated at commit time.

Setting constraints ON causes each of the affected constraints to be evaluated immediately, as well as at the end of each statement, until the SET ALL CONSTRAINTS OFF statement is issued or until the transaction completes with a commit or rollback operation.

The SET ALL CONSTRAINTS statement overrides the CONSTRAINT_MODE=IMMEDIATE or the CONSTRAINT_MODE=DEFERRED qualifier.

SQL users who require ANSI-standard SQL compatibility should set constraints IMMEDIATE. The default (CONSTRAINT_MODE=DEFERRED) is acceptable for most other users.

You can use the ON keyword instead of IMMEDIATE and the OFF keyword instead of DEFERRED.

CONTEXT=

Instructs the SQL module processor to execute module language procedures in the context of a particular distributed transaction. When you use this qualifier, SQL generates an additional parameter for the procedures and places the parameter as the last parameter declared in the procedure.

Following are the options you can specify with the CONTEXT= qualifier:

- NONE

  Specifies that the SQL module processor does not add a context parameter to any procedure in the module.

- ALL

  Specifies that the SQL module processor adds a context parameter to every procedure in the module.

- procedure-list

  Specifies that the SQL module processor adds a context parameter to each procedure listed. If you specify an entry name for a procedure in the list, the SQL module processor changes the name of that procedure to the name specified.

  For example, you can specify the following qualifier on the command line:

  ```
  /CONTEXT=(OPEN_PROC :OPEN_PROC_DIST, FETCH_PROC :FETCH_PROC_DIST,-
           CLOS_PROC :CLOS_PROC_DIST)
  ```

  SQL passes the context parameter to the OPEN_PROC, FETCH_PROC, and CLOS_PROC procedures and gives them the new names specified. For more information, see the *Oracle Rdb7 Guide to Distributed Transactions*.

Your application must use the context structure to pass the address of the distributed TID from the host language program to procedures in the module that are involved in the distributed transaction. You pass the context structure to procedures that contain executable SQL statements, except statements that you cannot execute when a transaction is already started or statements that you cannot use when you explicitly call the DECdtm system services. Section 2.9 lists the nonexecutable statements that do not take a context structure.

You can also use the CONTEXT qualifier to specify a new name for a procedure.

Qualifiers used with the CONTEXT qualifier specify which procedures receive context parameters, and whether or not the names of the procedures are changed.

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. Because batch-update transactions do not write to recovery-unit journal (.ruj) files, batch-update transactions cannot be rolled back.)

If you attempt to start a distributed transaction using a batch-update transaction, what happens depends upon whether you call the DECdtm system services implicitly or explicitly and which SQL statement you use to start the transaction:

- If you start a batch-update transaction and explicitly call the DECdtm system services, SQL returns an error at compile time.
- If you start a batch-update transaction and implicitly call the DECdtm system services, SQL takes the following actions:
  - If you use a SET TRANSACTION statement with the BATCH UPDATE clause, SQL starts a nondistributed transaction.
  - If you use a DECLARE TRANSACTION statement with the BATCH UPDATE clause, SQL returns an error at compile time.

The two-phase commit protocol applies only to distributed transactions. For more information about distributed transactions, see the *Oracle Rdb7 Guide to Distributed Transactions*.

USER_DEFAULT=username
Specifies the user name at compile time.

If you use the USER DEFAULT clause of the DECLARE ALIAS statement, you use this qualifier to pass the compile-time user name to the program.

PASSWORD_DEFAULT=password
Specifies the user's password at compile time.

If you use the USING DEFAULT clause of the DECLARE ALIAS statement, you use this qualifier to pass the compile-time user's password to the program.

module-qualifiers-2
A set of qualifiers that you can optionally apply to the SQL module processor command line.

database-options
Specifies that the SQL module processor will process a program for access to the specified database type.

For more information regarding database options, see Section 2.10.

OPTIMIZATION_LEVEL optimization_option
Specifies the optimizer strategy to be used to process all queries within your SQL module language program. Select the:

- 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. strategy.

- 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 indices.
- 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.

You can select either the TOTAL_TIME or the FAST_FIRST option in conjunction with either the AGGRESSIVE_SELECTIVITY or SAMPLED_SELECTIVITY option. Use a comma to separate the keywords and enclosed the list in parentheses.

The following example shows how to use the OPTIMIZATION_LEVEL qualifier:

```
$ SQL$MOD/OPTIMIZATION_LEVEL=(TOTAL_TIME,SAMPLED_SELECTIVITY) APPCODE.SQLMOD
```

Any query that explicitly includes an OPTIMIZE WITH, or OPTIMIZE_FOR clause is not affected by the settings established using the OPTIMIZATION_LEVEL qualifier.

You affect the optimizer strategy of *static* SQL queries with the optimization level qualifier; however, the default optimizer strategy set by the OPTIMIZATION_LEVEL qualifier can be overridden by the default optimizer strategy set in a top-level SELECT statement.

In contrast, the SET OPTIMIZATION LEVEL statement specifies the query optimization level for *dynamic* SQL query compilation only; the statement does not affect the SQL compile-time environment nor does it affect the run-time environment of static queries.

QUERY_TIME_LIMIT=total-seconds
Limits the number of records returned during query processing by counting the number of seconds used to process the query and returning an error message if the query exceeds the total number of seconds specified.

The default is unlimited time for the query to compile. Dynamic SQL options are inherited from the compilation qualifier.

QUERY_MAX_ROWS=total-rows
Limits the number of records returned during query processing by counting the number of rows returned by the query and returning an error message if the query exceeds the total number of rows specified.

The default is an unlimited number of record fetches. Dynamic SQL options are inherited from the compilation qualifier.

QUERY_CPU_TIME_LIMIT=total-seconds
Limits 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 time for the query to compile. Dynamic SQL options are inherited from the compilation qualifier.

ROLLBACK_ON_EXIT
Rolls back outstanding transactions when a program exits from SQL.

On OpenVMS outstanding transactions are committed when a program exits from SQL by default. Therefore, if you want to roll back changes, specify this qualifier on the command line.

Usage Notes

- Although SQL module language processes dynamic strings correctly in other contexts, you must use either a static descriptor or a dynamic descriptor of exact length whenever you use the GENERAL language.

  When you create an SQL module language source file specifying the GENERAL language and pass character parameters by descriptor, the length of the character string you pass must be equal to the maximum size of the character parameter specified. If you do not, SQL stores extraneous characters in the data-field character positions after those in the original dynamic string, instead of padding the string with blank spaces. The restriction applies only when you call the module language from a language that uses dynamic instead of static string descriptors (such as BASIC or VAX SCAN), in particular when passing a parameter to an INSERT operation.

  You can prevent this problem from occurring in a language such as BASIC by putting the string definition in a MAP declaration, which makes the string static instead of dynamic.

- Give different file names to host language module source files and their corresponding SQL module files, even though they have different file extensions. Both the SQL module processor and the host language compiler produce .obj files. If the source file names are not distinct, the LINK command may fail.

- When you compile an SQL module using Ada as the source language, Oracle Rdb generates an Ada package that contains Ada declarations for all procedures in the SQL module. Part of the declaration for each routine is the declaration of each parameter. These parameters will be declared using data types in the SQL_STANDARD package. Refer to the Usage Note at the end of Section 3.5 for further information.
- Using the database-options qualifier does not create backward compatilility. In this case, backward compatibility refers to accessing an older version of the database system remotely. However, the interface still needs support routines which are available only in the latest version. In other words, you cannot compile and link under Oracle Rdb V6.0 and run the program under Oracle Rdb V5.1.

Example

Example 1: Compiling and linking a program with an SQL module

The following example shows the commands to compile, link, and run the sample Pascal program in the example from Section 3.2:

```
$ SQLMOD :== $SQL$MOD
$ SQLMOD LIST_EMP_PASMOD.SQLMOD
$ PASCAL LIST_EMP.PAS
$ ! This LINK command requires that the logical name
$ ! LNK$LIBRARY is defined as SYS$LIBRARY:SQL$USER.OLB
$ LINK LIST_EMP.OBJ, LIST_EMP_PASMOD.OBJ
$ RUN LIST_EMP.EXE
Matching Employees:
Alvin     Toliver
Louis     Tarbassian
```

4

SQL Precompiler

The SQL precompiler lets you embed SQL statements directly in programs written in Ada, C, COBOL, FORTRAN, Pascal, and PL/I. In contrast, SQL module language allows procedures that contain SQL statements to be called from any host language. The SQL precompiler supports only specific languages. Chapter 3 describes the advantages of SQL module language as compared with the SQL precompiler.

For a detailed discussion of programming considerations when using the SQL precompiler, see the *Oracle Rdb Guide to SQL Programming*.

4.1 Embedding SQL Statements in Programs

You have a number of factors to consider when embedding SQL statements in a host language program. In the following sections, you learn how to use the two-phase commit protocol and how to embed clauses in the DECLARE MODULE statement to specify character sets, quoting rules, default date format, and so forth.

4.1.1 Embedding Module Clauses in Host Language Code

You can include module clauses in a DECLARE MODULE statement in your host language programs to control:

- Dialect settings, which let you specify with one clause: character length, double quotation marks, identifiers as keywords, read-only views, and the interpretation of DATE and CURRENT_TIMESTAMP data types
- Character sets, which specify the literal, national, default, identifier and display character sets for the module
- Schema name, which names the default schema name for the module
- Authorization identifier, which specifies the authorization identifier for the module

- Module language options, which specify the alias for the module, individual dialect settings (character length, quoting rules, and so forth), colons for prefixing parameter names, and privilege checking for executing a module

For more information about using module clauses in the DECLARE MODULE statement, see the DECLARE MODULE Statement.

4.1.2 Using the Two-Phase Commit Protocol in Embedded Programs

When you use precompiled SQL, you can explicitly use the two-phase commit protocol. To do this, your application must explicitly call the transaction manager and declare a context structure. The context structure contains the distributed transaction identifier (TID) as one of its elements. In addition, most executable SQL statements involved in the distributed transaction must include a USING CONTEXT clause. The USING CONTEXT clause associates the context structure with the SQL statement. Section 2.9 lists the nonexecutable statements that do not take a context structure.

The following syntax diagram shows the format for an embedded SQL statement that is part of a distributed transaction:

EXEC SQL → USING CONTEXT → <:variable> → simple-statement

For example, the following embedded SQL statement opens a cursor as part of a distributed transaction:

```
EXEC SQL USING CONTEXT :DISTR_TRANS OPEN CURSOR1
```

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. Because batch-update transactions do not write to recovery-unit journal (.ruj) files, batch-update transactions cannot be rolled back.)

If you attempt to start a distributed transaction using a batch-update transaction, what happens depends upon whether you call the DECdtm system services implicitly or explicitly, and which SQL statement you use to start the transaction:

- If you start a batch-update transaction and explicitly call the DECdtm system services, SQL returns an error at compile time.

- If you start a batch-update transaction and implicitly call the DECdtm system services, SQL takes the following actions:
  - If you use a SET TRANSACTION statement with the BATCH UPDATE clause, SQL starts a nondistributed transaction.
  - If you use a DECLARE TRANSACTION statement with the BATCH UPDATE clause, SQL returns an error at compile time.

The two-phase commit protocol applies only to distributed transactions. For more information about distributed transactions, see the *Oracle Rdb7 Guide to Distributed Transactions*.

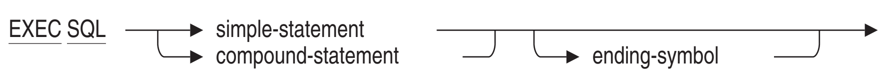
4.2 SQL Precompiler Syntax

The SQL precompiler provides special keywords and syntax that allow you to include (embed) simple and compound statements directly into host language programs. Then you can use the SQL precompiler to process the combined embedded statements and host language code to produce an object file for linking and execution.

Environment

You can use SQL precompiler syntax only in Ada, C, COBOL, FORTRAN, Pascal, and PL/I host language source files. The SQL precompiler supports no other host languages. If you use a host language other than the ones mentioned for embedded SQL and you want to use the SQL interface with it, you must use the SQL module processor.

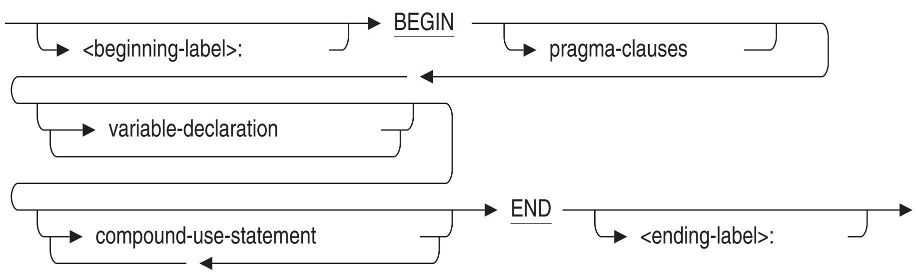
Format



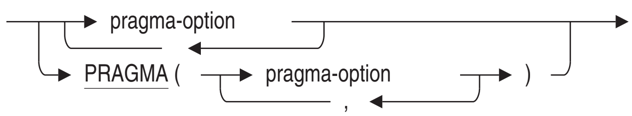
ending-symbol =

simple-statement =

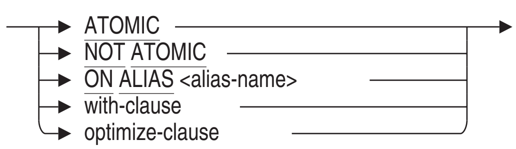
compound-statement =



pragma-clauses =

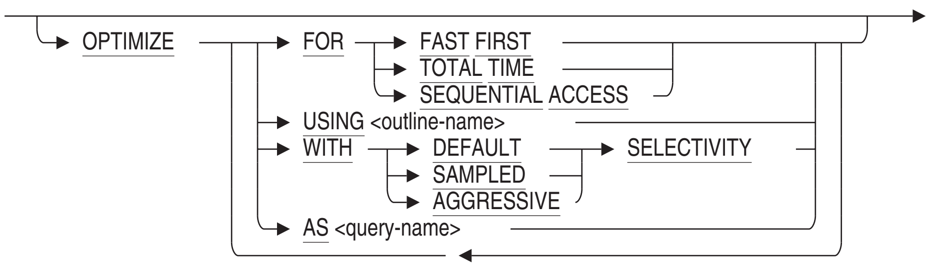


pragma-option =



with-clause =

optimize-clause =



Arguments

compound-statement

A statement that can contain multiple SQL statements in an SQL module procedure or in an embedded SQL procedure.

An embedded procedure that contains a compound statement is called an **embedded multistatement procedure**. SQL supports a restricted subset of SQL statements in a compound statement embedded in a host langauge program. Refer to Table 1–1 for a list of valid SQL statements allowed in a compound statement.

Compound statements can also include program-like, flow-of-control statements (IF, LOOP, CASE, LEAVE), transaction management statements (COMMIT and ROLLBACK), a variable declaration statement (SET assignment), a cursor-processing statement (FOR), and a procedure-debugging statement (TRACE).

See the Compound Statement for a complete description of a compound statement.

ending-symbol

Ends an embedded simple or an embedded compound statement. To end an embedded statement, follow the host language requirements listed in Table 4–1.

Table 4–1 Ending Embedded SQL Statements

Language	Symbols to End EXEC SQL Statements
Ada	Semicolon (;)
C	Semicolon (;)
COBOL	END-EXEC
FORTRAN	Ending symbol not required
Pascal	Semicolon (;)
PL/I	Semicolon (;)

EXEC SQL
Prefixes each simple or compound statement. Converting interactive statements to precompiled statements requires the added step of starting each simple or compound statement with the keywords EXEC SQL. SQL cannot process these statements otherwise. Also, both keywords EXEC and SQL must be on the same line, and you cannot insert comments between them.

simple-statement
A statement that can contain a single SQL statement only. Refer to Table 1–1 for a list of SQL statements that are valid within a simple statement.

See the Simple Statement for a complete description of a simple statement.

Usage Notes

- An embedded compound statement cannot include either a beginning or an ending label.
- The keyword PROCEDURE cannot be used in an embedded SQL procedure.
- If the embedded statement is a compound statement, the local variable can conceal a host variable of the same name for the duration of the BEGIN . . . END block.
- If a DECLARE TABLE statement appears before a CREATE DATABASE statement, your compilation could fail with an error message indicating that SQL$DATABASE or SQL_DATABASE could not be opened or that certain database objects could not be found in your database.

The SQL precompiler processes metadata statements before other statements. If your DECLARE TABLE statement is found before the CREATE DATABASE statement that defines it, then SQL will try to attach to SQL$DATABASE or SQL_DATABASE for the metadata lookups.

Place your CREATE DATABASE statement before your DECLARE TABLE statements.

- The SQL Precompiler also supports the syntax BEGIN DECLARE SECTION. This clause is ambiguous because of the BEGIN DECLARE of the compound statement. Therefore, within the EXEC-SQL compound statement only one pragma clause can be selected. The use of the PRAGMA list allows all options to be specified.
- The clauses ON ALIAS, OPTIMIZE and WITH HOLD must only appear on the outer-most BEGIN of a compound statement.

Example

Example 1: Embedding a compound statement in a host language program

The following example shows how to embed a multistatement procedure in a program. The keyword PROCEDURE does not appear in an embedded SQL application.

```
EXEC SQL BEGIN DECLARE SECTION ;
   int   x ;
EXEC SQL END DECLARE SECTION ;

EXEC SQL
   BEGIN
      DECLARE :y INTEGER ;

      SET :y = 2 * :x ;  -- :x is a host variable

      UPDATE employees
      SET salary = :y ;
      WHERE .
            .
            . ;

         BEGIN
            DECLARE :x INTEGER ;

            SET :x = 100 ; -- :x is a local variable

            UPDATE employees
            SET salary = :x ;
            WHERE .
                  .
                  . ;
         END ;

   END ;
```

4.3 SQL Precompiler Command Line

You can define a symbol to help you invoke the SQL precompiler:

```
$ SQLPRE == "$SQL$PRE"
```

Because the SQL precompiler requires a language qualifier, you might want to define a particular language so that you can invoke the command on one line:

```
$ SADA == "$SQL$PRE/ADA"
$ SADA SQL_DYNAMIC
```

By defining symbols, you can invoke the SQL precompiler with or without a file specification for a host language program file:

- If you invoke the SQL precompiler without an input file specification for a host language program file, the precompiler prompts you for it. For example:

  ```
  $ SQLPRE
  INPUT FILE> pre-host-file-spec
  ```

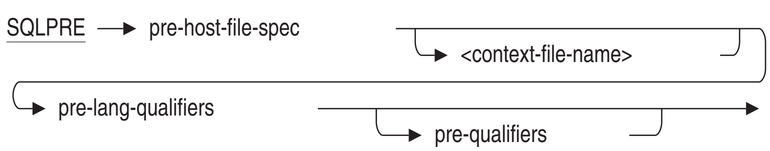
- If you invoke the SQL precompiler with a host language program file as part of the DCL command line, SQL starts processing your file immediately after you press the Return key. For example:

  ```
  $ SADA pre-host-file-spec pre-qualifiers
  ```

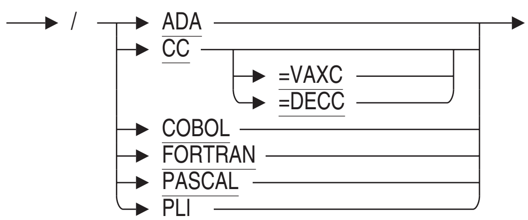
Whichever method you choose to invoke the precompiler, you have the option to specify a wide range of qualifiers that control how the SQL precompiler processes the module file. The syntax diagrams show the format for the qualifiers that you can include with the host language program file specification.

Format

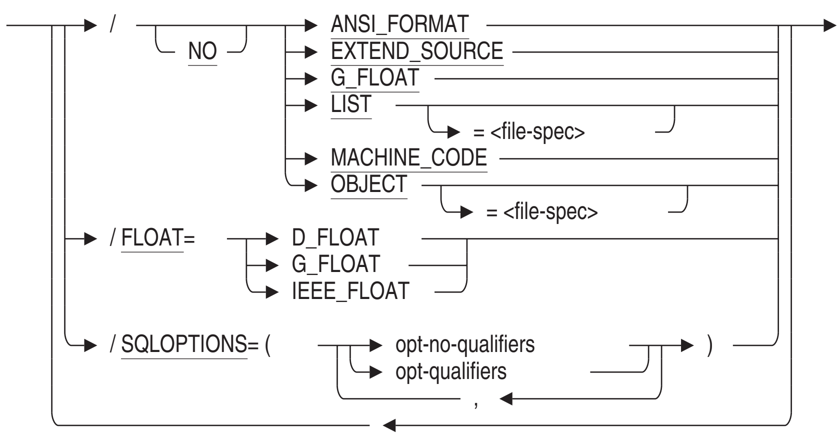
pre-host-file-qual =



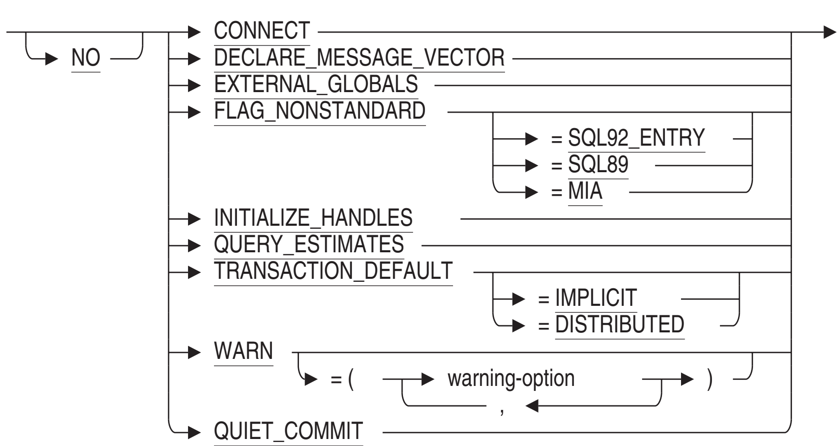
pre-lang-qualifiers =



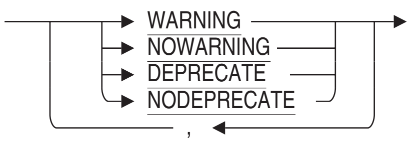
pre-qualifiers =



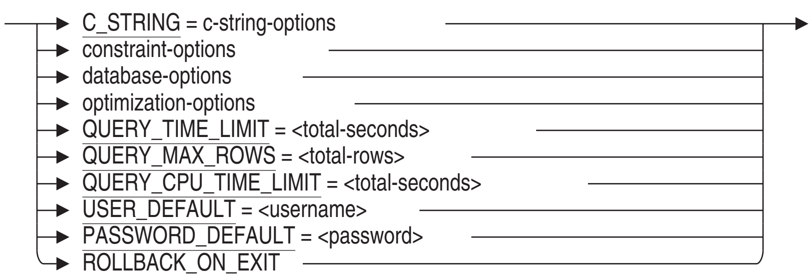
opt-no-qualifiers =



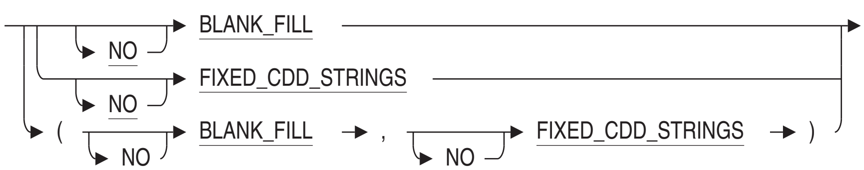
warning-option =



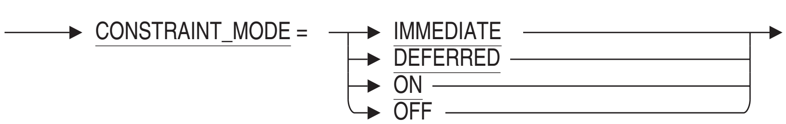
opt-qualifiers =



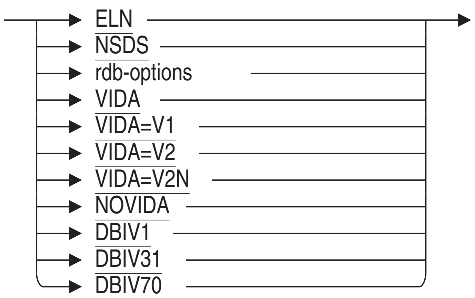
c-string-options =



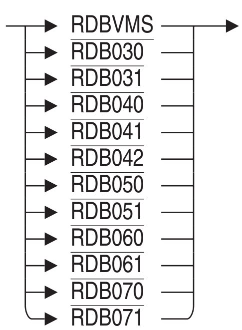
constraint-options =



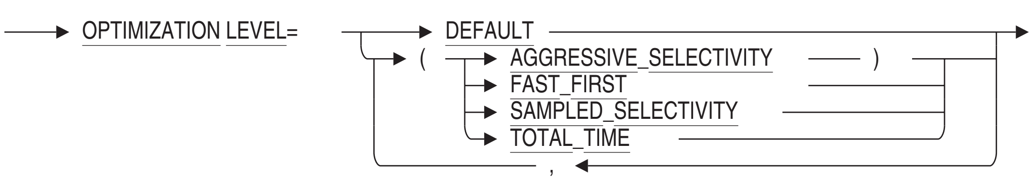
database-options =



rdb-options =



optimization-options=



Arguments

ANSI_FORMAT
NOANSI_FORMAT
Specifies whether the SQL precompiler accepts terminal-format COBOL or ANSI-format COBOL.

The default is the terminal format COBOL NOANSI_FORMAT qualifier.

c-string-options
Controls how SQL handles C host language character strings.

Use either or both of the [NO]BLANK_FILL and [NO]FIXED_CDD_STRINGS options with the C_STRING keyword to control C string characteristics.

SQLOPTIONS= (C_STRING = [NO]BLANK_FILL)
SQLOPTIONS= (C_STRING = [NO]FIXED_CDD_STRINGS)
SQLOPTIONS= (C_STRING = ([NO]BLANK_FILL, [NO]FIXED_CDD_STRINGS))
Specifies how to handle C host language character strings:

- [NO]BLANK_FILL (default: BLANK_FILL)

  Controls whether or not C character strings are filled with blanks as required by the SQL89 and ANSI/ISO SQL standards or if the null terminator is placed after the last data byte of the source string.

- [NO]FIXED_CDD_STRINGS (default: NOFIXED_CDD_STRINGS)

  Controls whether or not SQL treats C character strings from CDD/Repository record definitions as fixed-length character strings or C null-terminated strings.

SQLOPTIONS= (CONNECT)
SQLOPTIONS= (NOCONNECT)
Specifies whether or not SQL allows multiple user connections and access to global databases across modules. All SQL modules in an application must be compiled with connections enabled or disabled.

The SQLOPTIONS=NOCONNECT qualifier is the default.

SQLOPTIONS= (CONSTRAINT_MODE=IMMEDIATE)
SQLOPTIONS= (CONSTRAINT_MODE=DEFERRED)
You can optionally specify either the SQLOPTIONS=(CONSTRAINT_MODE=IMMEDIATE) or SQLOPTIONS=(CONSTRAINT_MODE=DEFERRED) qualifier on the SQL precompiler command line to set the default constraint evaluation mode for commit-time constraints. (This qualifier does not affect the evaluation of verb-time constraints.) The default is DEFERRED; that is, commit-time constraints are evaluated at commit time.

Setting constraints IMMEDIATE causes each affected constraint to be evaluated immediately, as well as at the end of each statement, until the SET ALL CONSTRAINTS DEFERRED statement is issued, or until the transaction completes with a commit or rollback operation.

The SET ALL CONSTRAINTS statement overrides the constraint evaluation mode specified in the SQLOPTIONS qualifier. For more information about the default constraint mode, see SET Statement.

SQL users who require ANSI-standard SQL compatibility should set constraints as IMMEDIATE. The default (CONSTRAINT_MODE=DEFERRED) is acceptable for most other users.

SQLOPTIONS= (CONSTRAINT_MODE=ON)
SQLOPTIONS= (CONSTRAINT_MODE=OFF)
The qualifiers CONSTRAINT_MODE=ON and CONSTRAINT_MODE=OFF duplicate the behavior of the qualifiers CONSTRAINT_MODE=IMMEDIATE and CONSTRAINT_MODE=DEFERRED, respectively.

context-file-name
An SQL command procedure containing DECLARE statements that you want to apply when your program compiles and executes. See Section 2.11 for information about context-file-name.

database-options
Specifies that the SQL precompiler correctly processes a program for access to the specified database type. For more information regarding database options, see Section 2.10.

The precompiler database option can in turn be overridden by an attach to a database at run time. On the DECLARE statement, SQL sets the database options of the specified database.

By default, the SQL precompiler determines the valid database from the database used to compile the program. If no database is used to compile the program, the precompiler processes the program for a database created with the most recent version of Oracle Rdb.

SQLOPTIONS= (DECLARE_MESSAGE_VECTOR)
SQLOPTIONS= (NODECLARE_MESSAGE_VECTOR)
Specifies that the RDB$MESSAGE_VECTOR structure be declared in the host language as part of the SQLCA during SQLPRE processing. You can use this switch with language compilers that support the '$' special character.

The default is the SQLOPTIONS=(DECLARE_MESSAGE_VECTOR) qualifier.

EXTEND_SOURCE
NOEXTEND_SOURCE
Allows the SQL precompiler to view 132 columns of FORTRAN source rather than the default of 72 columns.

SQLOPTIONS= (EXTERNAL_GLOBALS)
SQLOPTIONS= (NOEXTERNAL_GLOBALS)
Specifies whether or not alias references are coerced into alias definitions. An **alias definition** is an alias declared with the GLOBAL keyword (the default) in the DECLARE ALIAS statement. An **alias reference** is an alias declared with the EXTERNAL keyword in the DECLARE ALIAS statement.

The EXTERNAL_GLOBALS qualifier treats alias references as alias definitions. This qualifier provides compatibility with versions prior to V7.0.

The NOEXTERNAL_GLOBALS qualifier treats alias references as alias references. The NOEXTERNAL_GLOBALS qualifier may be useful on OpenVMS if your application shares an alias between multiple shareable images.

The default on OpenVMS is the SQLOPTIONS=(EXTERNAL_GLOBALS) qualifier.

See the DECLARE ALIAS Statement for more information about alias definitions and references. For information on using aliases and shareable images, see the *Oracle Rdb Guide to SQL Programming*.

SQLOPTIONS= (FLAG_NONSTANDARD)
SQLOPTIONS= (FLAG_NONSTANDARD =SQL92_ENTRY)
SQLOPTIONS= (FLAG_NONSTANDARD =SQL89)
SQLOPTIONS= (FLAG_NONSTANDARD =MIA)
SQLOPTIONS= (NOFLAG_NONSTANDARD)
Specifies whether or not SQL identifies nonstandard syntax. Nonstandard syntax, called an extension, refers to syntax that is not part of the ANSI/ISO SQL standard or the Multivendor Integration Architecture (MIA) standard. You can specify the following options:

- (FLAG_NONSTANDARD)

  Notifies you of syntax that is an extension to the ANSI/ISO SQL standard.

- (FLAG_NONSTANDARD=SQL92_ENTRY)

  Notifies you of syntax that is an extension to the ANSI/ISO SQL standard. This qualifier has the same effect on flagging as does the (FLAG_NONSTANDARD) qualifier.

- (FLAG_NONSTANDARD=SQL89)

  Notifies you of syntax that is an extension to the ANSI/ISO 1989 standard.

- (FLAG_NONSTANDARD=MIA)

  Notifies you of syntax that is an extension to the MIA standard.

- (NOFLAG_NONSTANDARD)

  Prevents notification of extensions.

The default is the SQLOPTIONS=(NOFLAG_NONSTANDARD) qualifier.

FLOAT=D_FLOAT
FLOAT=G_FLOAT
FLOAT=IEEE_FLOAT

Specifies the floating point representation that the SQL precompiler uses for floating point data types in a formal parameter list and specifies the floating point qualifier passed to the language compiler.

The SQL Precompiler translates embedded SQL into host language declarations and procedure calls. In addition it generates the procedures behind the procedure calls. The /FLOAT qualifier for SQL$PRE determines the floating point format that SQL$PRE assumes for host language variables and, hence, determines the conversions that will be made internal to the generated SQL procedures. When SQL$PRE calls the host language compiler to process the precompiled program it passes an equivalent qualifier to its /FLOAT qualifer that is supported by the host language. This means that to the extent that the floating point format of host language variables is determined by a /FLOAT qualifier, the floating point formats of the host language variables and the parameters of procedure calls generated by SQL$PRE are guaranteed to be compatible. When the host language provides a type which explicitly declares the floating point format of the an individual variable, SQL$PRE uses that information to determine the conversion needed regardless of the setting of the /FLOAT qualifier.

The SQL Precompiler's default floating point format for single or double precision floating point types is F-Floating and G-Floating format, respectively. This is equivalent to using a qualifier of /FLOAT=G_FLOAT with the SQL$PRE command.

If a host language variable is a record or structure (for example a qualified parameter in the INTO clause of a singleton SELECT statement), any fields in the record or structure that are of a floating point type follow the same rules as described above.

There are a few cases where a host language provides mechanisms for specifying floating point format that are not recognized by SQL$PRE. In these cases, it is the developer's responsibility to ensure that the format is what SQL$PRE expects. These cases are described in host language-specific sections in Section 4.4.

Note

The Common Data Dictionary supports floating point types. However when the /FLOAT qualifier specifies IEEE_FLOAT, these types may not be used.

G_FLOAT
NOG_FLOAT
The /G_FLOAT and /NOG_FLOAT qualifiers are for backwards compatibility. They are equivalent to /FLOAT=G_FLOAT and /FLOAT=D_FLOAT, respectively. You should not specify both /FLOAT and /[NO]G_FLOAT qualifiers.

SQLOPTIONS= (INITIALIZE_HANDLES)
SQLOPTIONS= (NOINITIALIZE_HANDLES)
Specifies whether or not alias definitions are coerced into alias references. The NOINITIALIZE_HANDLES qualifier causes all alias declarations to be treated as alias references.

An **alias definition** is an alias declared with the GLOBAL keyword (the default) in the DECLARE ALIAS statement. An **alias reference** is an alias declared with the EXTERNAL keyword in the DECLARE ALIAS statement.

The NOINITIALIZE_HANDLES qualifier may be useful for existing source code on OpenVMS in coercing alias definitions into alias references. Because there is usually no distinction between a definition and a reference on OpenVMS, your application might declare an alias definition where an alias reference is needed. If you reorganize your application into multiple images that share aliases, you must distinguish the alias definition from the alias reference. In this case, use the NOINITIALIZE_HANDLES qualifier to coerce a definition into a reference without changing your source code.

If your application correctly declares alias references with the EXTERNAL keyword, use the NOEXTERNAL_GLOBALS qualifier, instead of [NO]INITIALIZE_HANDLES to override the default on OpenVMS and cause SQL to treat alias references properly as references.

The default is the SQLOPTIONS=INITIALIZE_HANDLES qualifier. This qualifier overrides the [NO]EXTERNAL_GLOBALS qualifier.

The SQLOPTIONS=[NO]INITIALIZE_HANDLES qualifier is maintained for compatibility with previous versions of Oracle Rdb. For V7.0 and higher, use the [NO]EXTERNAL_GLOBALS qualifier, which provides more precise control over alias definition.

See the DECLARE ALIAS Statement for more information about alias definitions and references. For information on using aliases and shareable images, see the *Oracle Rdb Guide to SQL Programming*.

LIST
NOLIST
With Oracle Rdb for OpenVMS Alpha, determines whether or not the SQL precompiler generates a list file (default file extension .lis) that contains information about the SQL compilation and the host language compilation. In addition, if the logical name SQL$KEEP_PREP_FILES is defined, the SQL precompiler retains an intermediate module list file (file extension .mli), which contains information about the SQL compilation only. If you specify the LIST qualifier and do not include a file specification, the SQL precompiler creates a list file with the same file name as your source file with the file extension .lis.

The NOLIST qualifier is the default.

MACHINE_CODE
NOMACHINE_CODE
With Oracle Rdb for OpenVMS Alpha, specifies whether or not the SQL precompiler includes machine code in the list file; however, to generate the list file with the machine code in it, you must also specify the LIST qualifier.

The NOMACHINE_CODE qualifier is the default.

OBJECT
NOOBJECT
Specifies whether or not the SQL precompiler creates an object file when compiling the source file if the compilation does not generate fatal errors; and, if an object file is produced, what the file is named. If you specify the OBJECT qualifier and do not include a file specification, the precompiler creates an object file with the same file name as the source file and the file extension .obj. You can specify the OBJECT qualifier for any language except Ada.

The OBJECT qualifier is the default.

OPTIMIZATION_LEVEL=optimization_options
Specifies the optimizer strategy to be used to process all queries within your SQL module language program. Select the:

- 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 indices.
- 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.

You can select either the TOTAL_TIME or the FAST_FIRST option in conjunction with either the AGGRESSIVE_SELECTIVITY or SAMPLED_SELECTIVITY option. Use a comma to separate the keywords and enclosed the list in parentheses.

The following example shows how to use the OPTIMIZATION_LEVEL qualifier:

```
$ SQL$PRE/SQLOPTIONS=OPTIMIZATION_LEVEL=(TOTAL_TIME,SAMPLED_SELECTIVITY) APPCODE.SC
```

Any query that explicitly includes an OPTIMIZE WITH, or OPTIMIZE_FOR clause is not affected by the settings established using the OPTIMIZATION_LEVEL qualifier.

You affect the optimizer strategy of *static* SQL queries with the optimization level qualifier; however, the default optimizer strategy set by the OPTIMIZATION_LEVEL option can be overridden by the default optimizer strategy set in a top-level SELECT statement.

In contrast, the SET OPTIMIZATION LEVEL statement specifies the query optimization level for *dynamic* SQL query compilation only; the statement does not affect the SQL compile-time environment nor does it affect the run-time environment of static queries.

optimization-options
Specifies the optimizer strategy to be used for processing all queries within your SQL precompiler program.

SQLOPTIONS= (PASSWORD_DEFAULT=password)
Specifies the user's password at compile time.

If you use the USING DEFAULT clause of the DECLARE ALIAS statement, you use this qualifier to pass the compile-time user's password to the program.

pre-host-file-spec

The file specification for a host language source file that contains embedded SQL statements. The default file extension for the source file depends on the host language specified in the language qualifier.

Language	Default File Extension
Ada	.sqlada
C	.sc
COBOL	.sco
FORTRAN	.sfo
Pascal	.spa
PL/I	.spl

If the host language is Ada or COBOL, the file name (without the file extension) cannot be longer than 27 characters.

The precompiler command line allows a list of host language source files in this argument, but only processes the first file specification it encounters. If you specify a list of files, the precompiler:

- Gives a warning message that only the first file on the line will be precompiled
- Ignores the other file specifications and passes them along to the host language compiler in the same order as they appeared on the precompiler command line

For instance, the following command lines are valid, but only the MY_FILE host language file is precompiled:

```
$ SQLPRE/PLI/LIS/DEB MY_FILE+MY_TLB_1/LIB+MY_TLB_2/LIB
$ SQLPRE/PASCAL MY_FILE,MY_OTHER_FILE
$ SQLPRE/COB/DEB MY_FILE,MY_NODB_FILE
$ SQLPRE/CC MY_FILE+REST_OF_APPL+APPL_TLB/LIB
```

For the previous command lines, the precompiler passes the following corresponding command lines to the host language compiler:

```
$ PLI/LIS/DEB MY_FILE.PLI;n+MY_TLB_1/LIB+MY_TLB_2/LIB/NOG_FLOAT
$ PAS MY_FILE.PAS;n,MY_OTHER_FILE
$ COB/DEB MY_FILE.COB;n,MY_NODB_FILE
$ CC MY_FILE.C;n+REST_OF_APPL+APPL_TLB/LIB/NOG_FLOAT
```

The *;n* notation signifies the version number of the host language file generated by the SQL precompiler.

pre-lang-qualifiers
Refers to the host language in which the program containing embedded SQL procedures is written. You must supply a language qualifier. The host language qualifier values are ADA, CC, CC=VAXC, CC=DECC, COBOL, FORTRAN, PASCAL, and PLI.

The following statements apply to the CC SQL precompiler switch:

- The CC=VAXC switch instructs the precompiler to compile the source as a VAXC source. If the VAXC compiler is not installed, the DECC compiler is used with the /STANDARD=VAXC host language compiler switch.
- The CC=DECC switch instructs the precompiler to compile the source as a DECC source. If the DECC compiler is not installed, you will get a DCL error.
- The default keyword, either VAXC or DECC, is based on your system configuration. If the VAXC compiler is installed on your system, VAXC is the default keyword. If the DECC compiler is installed, DECC is the default keyword. If both compilers are installed, the default is based on whichever C compiler your system manager has specified.

pre-qualifiers
Refers to the optional qualifiers allowed on the SQL precompiler command line.

SQLOPTIONS= (QUERY_CPU_TIME_LIMIT=total-seconds)
Limits 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 time for the query to compile. Dynamic SQL options are inherited from the compilation qualifier.

SQLOPTIONS= (QUERY_ESTIMATES)
SQLOPTIONS= (NOQUERY_ESTIMATES)
Specifies whether or not SQL returns the estimated number of rows and estimated number of disk I/O operations in the SQLCA structure. If you specify the QUERY_ESTIMATES keyword, SQL returns the estimated number of rows in the field SQLCA.SQLERRD[2] and the estimated number of disk I/O operations in the field SQLCA.SQLERRD[3]. The value of SQLCA.SQLERRD[2] and SQLCA.SQLERRD[3] is normally 0 after you execute an OPEN statement for a table.

The SQLOPTIONS=QUERY_ESTIMATES qualifier is the default.

SQLOPTIONS= (QUERY_MAX_ROWS=total-rows)

Limits the number of records returned during query processing by counting the number of rows returned by the query and returning an error message if the query exceeds the total number of rows specified.

The default is an unlimited number of record fetches. Dynamic SQL options are inherited from the compilation qualifier.

SQLOPTIONS= (QUERY_TIME_LIMIT=total-seconds)

Limits the number of records returned during query processing by counting the number of seconds used to process the query and returning an error message if the query exceeds the total number of seconds specified.

The default is unlimited time for the query to compile. Dynamic SQL options are inherited from the compilation qualifier.

QUIET COMMIT ON
QUIET COMMIT OFF

The QUIET COMMIT ON clause disables error reporting for the COMMIT and ROLLBACK statements if either statement is executed when no transaction is active.

By default, if there is no active transaction, SQL will raise an error when COMMIT or ROLLBACK is executed. This default is retained for backward compatibility for applications that wish to detect the situation. If QUIET COMMIT is set to ON, a COMMIT or ROLLBACK executes successfully when there is no active transaction.

Note

Within a compound statement, the COMMIT and ROLLBACK statements are always ignored if no transaction is active.

SQLOPTIONS= (ROLLBACK_ON_EXIT)

Rolls back outstanding transactions when a program exits from SQL.

On OpenVMS, outstanding transactions are committed when a program exits from SQL by default. Therefore, if you want to roll back changes, specify this qualifier on the command line.

SQLOPTIONS= (TRANSACTION_DEFAULT = IMPLICIT)
SQLOPTIONS= (TRANSACTION_DEFAULT = DISTRIBUTED)

SQLOPTIONS= (NOTRANSACTION_DEFAULT)
Specifies when SQL starts a transaction and how SQL handles default distributed transactions. You can specify the following options:

- SQLOPTIONS = (TRANSACTION_DEFAULT = IMPLICIT)

  Causes SQL to start a transaction when you issue either a SET TRANSACTION statement or the first executable SQL statement in a session.

- SQLOPTIONS = (TRANSACTION_DEFAULT = DISTRIBUTED)

  Causes SQL to use the distributed transaction identifier (TID) for the default distributed transaction established by the DECdtm system service SYS$START_TRANS. Using this option eliminates the need to declare context structures in SQL precompiled programs and to use the USING CONTEXT clause in embedded SQL statements. Because it closes all cursors, it also eliminates the need to call the SQL_CLOSE_CURSORS routine.

  You must explicitly call the DECdtm system services when you use this option.

  This option provides support for the Structured Transaction Definition Language (STDL) of the Multivendor Integration Architecture (MIA) standard.

  If you specify the USING CONTEXT clause in embedded SQL statements, you must declare a context structure.

- SQLOPTIONS=(NOTRANSACTION_DEFAULT)

  Causes SQL not to start a transaction unless you execute a SET TRANSACTION statement. If you use this qualifier and issue an executable statement without first issuing a SET TRANSACTION statement, SQL returns an error.

The default is SQLOPTIONS = (TRANSACTION_DEFAULT = IMPLICIT).

SQLOPTIONS= (USER_DEFAULT=username)
Specifies the user name at compile time.

If you use the USER DEFAULT clause of the DECLARE ALIAS statement, you use this qualifier to pass the compile-time user name to the program.

SQLOPTIONS= WARN
SQLOPTIONS= NOWARN
Specifies whether or not the SQL precompiler writes informational and warning messages to the preprocessed host language source file and to

SYS$ERROR and SYS$OUTPUT (if different from SYS$ERROR). The WARN qualifier accepts the following options:

- [NO]WARNING

  Specifies whether or not the SQL precompiler writes informational and warning messages to your terminal, a list file, or both.

- [NO]DEPRECATE

  Specifies whether or not the SQL precompiler writes diagnostic messages about deprecated features.

  Deprecated features are features that are currently allowed but will not be allowed in future versions of SQL; that is, they will be obsolete. For example, one deprecated feature is the use of obsolete keywords such as VERB_TIME instead of VERB TIME. A complete list of deprecated features appears on line in the interactive SQL Help utility.

The SQLOPTIONS=WARN qualifier is equivalent to the SQLOPTIONS=WARN=(WARNING, DEPREC qualifier. The SQLOPTIONS=NOWARN qualifier is equivalent to the SQLOPTIONS=WARN=(NOWARNING, NODEPRECATE) qualifier.

You can specify the SQLOPTIONS=WARN=WARNING qualifier if you prefer to have all warning messages except those about deprecated features. You can specify the SQLOPTIONS=WARN=(NOWARNING, DEPRECATE) qualifier if you prefer the deprecated feature messages *only*.

warning-option
Specifies whether the SQL precompiler writes warning or diagnostic messages to your terminal, a list file, or both. Use either or both the [NO]WARNING or [NO]DEPRECATE options with the WARN qualifier. If you specify only a single warning option, you do not need to use parentheses.

Usage Notes

- Precompilers are restricted to 32767 bytes or less on the command line.

4.4 Host Language Variable Declarations Supported by the Precompiler

The SQL precompiler recognizes only a subset of valid host language variable declarations. If you refer to a variable declaration that SQL does not recognize in an embedded SQL statement, the precompiler generates a fatal error when it encounters that reference.

Oracle Rdb databases and the various host languages supported by the SQL precompiler do not necessarily support the same set of data types. The precompiler recognizes host language variable declarations that are equivalent to SQL data types plus a subset of other host language variable declarations.

- For host language variable declarations of data types that are equivalent to SQL data types, the precompiler passes values directly between the database and the host language variable.
- Host language floating point data types will be interpreted as having representations as determined by the /FLOAT qualifier on the precompiler command line and individual language rules. These rules are discussed in the host language-specific sections that follow. In these sections, selects will be shown from a table defined as follows:

  ```
  CREATE TABLE TESTTBL (
      KEYFIELD CHAR(10) PRIMARY KEY,
      FLOAT1   REAL,
      FLOAT2   DOUBLE PRECISION);
  ```

- For each host language, the precompiler also supports a limited number of host language variable declarations that do not correspond to SQL data types. SQL converts database values to the host language data type and host language values to the supported data type. SQL makes this conversion only for a subset of valid host language declarations.

  Table 4–2 shows the date-time data types that the precompiler supplies.

Table 4–2 Precompiler Date-Time Data Mapping

Module Language and Interactive SQL	Precompiler
DATE	SQL_DATE
DATE_ANSI	SQL_DATE_ANSI
DATE_VMS	SQL_DATE_VMS
TIME	SQL_TIME
TIMESTAMP	SQL_TIMESTAMP
INTERVAL YEAR	SQL_INTERVAL (YEAR)
INTERVAL YEAR TO MONTH	SQL_INTERVAL (YEAR TO MONTH)
INTERVAL MONTH	SQL_INTERVAL (MONTH)
INTERVAL DAY	SQL_INTERVAL (DAY)
INTERVAL DAY TO HOUR	SQL_INTERVAL (DAY TO HOUR)
INTERVAL DAY TO MINUTE	SQL_INTERVAL (DAY TO MINUTE)
INTERVAL DAY TO SECOND	SQL_INTERVAL (DAY TO SECOND)
INTERVAL HOUR	SQL_INTERVAL (HOUR)
INTERVAL HOUR TO MINUTE	SQL_INTERVAL (HOUR TO MINUTE)
INTERVAL HOUR TO SECOND	SQL_INTERVAL (HOUR TO SECOND)
INTERVAL MINUTE	SQL_INTERVAL (MINUTE)
INTERVAL MINUTE TO SECOND	SQL_INTERVAL (MINUTE TO SECOND)
INTERVAL SECOND	SQL_INTERVAL (SECOND)

- For all other host language variable declarations, the precompiler generates an error when it encounters a reference to them in embedded SQL statements.

The following sections list the subset of valid host language variable declarations that SQL recognizes. The sections also give examples of valid declarations that correspond to each of the SQL data types and examples of other declarations the precompiler does and does not recognize.

Note

The ANSI/ISO SQL standard specifies that variables used in embedded SQL statements must be declared within a pair of embedded SQL BEGIN DECLARE . . . END DECLARE statements. The Oracle Rdb SQL precompiler does not enforce this restriction. If you use the BEGIN DECLARE . . . END DECLARE statements, SQL generates a warning message when it encounters a variable declared outside of a BEGIN DECLARE . . . END DECLARE block.

If ANSI/ISO SQL compliance is important for your application, you should include all declarations for variables used in embedded SQL statements within a BEGIN DECLARE . . . END DECLARE block. See the BEGIN DECLARE Statement on the SQL module language for more information on the BEGIN DECLARE statement.

If you do not declare character variables using syntax that specifies a character set or by defining the RDB$CHARACTER_SET logical name, the SQL precompiler uses the UNSPECIFIED character set. When you use the UNSPECIFIED character set, the precompiler does not check to see if the character set of the variables matches the character sets of the database. For more information regarding the logical name, see Section 2.1.11.

The RDB$CHARACTER_SET logical name is deprecated and will not be supported in a future release.

The following sections do not discuss the requirements for declaring host language variables used as actual parameters in host language program calls to SQL module language procedures. Such host language variable declarations must correspond exactly to the corresponding formal parameter declarations in the SQL module file. If they do not, the program can generate unpredictable results at run time. See Chapter 3 for more information on the SQL module language.

4.4.1 Specifying Length of Character Parameters

To ensure that you specify the length of character variables correctly, use the following guidelines:

- For the C language, any character variables that correspond to character data type columns must be defined as the length of the longest valid column value in octets, plus 1 octet to allow for the null terminator.

- For other languages supported by the SQL precompiler, any character variables that correspond to character data type columns must be defined as the length of the longest valid column value in octets.
- When calculating the length of the longest valid column value, you must take into consideration whether the SQL precompiler interprets the length of columns in characters or octets. A program can control how the SQL precompiler interprets the length of columns in the following ways:
  - The CHARACTER LENGTH clause of the DECLARE MODULE statement
  - The DIALECT clause of the DECLARE MODULE statement
  - For dynamic SQL, the SET CHARACTER LENGTH statement

  See Table 2–2 for information about the number of octets used for one character in each character set.

Assume that you create the database MIA_CHAR_SET with the following character sets:

- Default character set: DEC_KANJI
- National character set: KANJI
- Identifier character set: DEC_KANJI

Assume that the database contains the table COLOURS and that the columns in that table are defined as shown in the following example:

```
SQL> SHOW DOMAINS;
User domains in database with filename MIA_CHAR_SET
ARABIC_DOM                      CHAR(8)
         ISOLATINARABIC 8 Characters,  8 Octets
DEC_KANJI_DOM                   CHAR(16)
GREEK_DOM                       CHAR(8)
         ISOLATINGREEK 8 Characters,  8 Octets
HINDI_DOM                       CHAR(8)
         DEVANAGARI 8 Characters,  8 Octets
KANJI_DOM                       CHAR(8)
         KANJI 4 Characters,  8 Octets
KATAKANA_DOM                    CHAR(8)
         KATAKANA 8 Characters,  8 Octets
MCS_DOM                         CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
RUSSIAN_DOM                     CHAR(8)
         ISOLATINCYRILLIC 8 Characters,  8 Octets
SQL> --
SQL> SHOW TABLE (COLUMNS) COLOURS;
Information for table COLOURS
```

```
Columns for table COLOURS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
ENGLISH                         CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
FRENCH                          CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
JAPANESE                        CHAR(8)          KANJI_DOM
         KANJI 4 Characters,  8 Octets
ROMAJI                          CHAR(16)         DEC_KANJI_DOM
KATAKANA                        CHAR(8)          KATAKANA_DOM
         KATAKANA 8 Characters,  8 Octets
HINDI                           CHAR(8)          HINDI_DOM
         DEVANAGARI 8 Characters,  8 Octets
GREEK                           CHAR(8)          GREEK_DOM
         ISOLATINGREEK 8 Characters,  8 Octets
ARABIC                          CHAR(8)          ARABIC_DOM
         ISOLATINARABIC 8 Characters,  8 Octets
RUSSIAN                         CHAR(8)          RUSSIAN_DOM
         ISOLATINCYRILLIC 8 Characters,  8 Octets
```

If your SQL precompiled program specifies CHARACTER LENGTH CHARACTERS, you would declare the corresponding variables as shown in the following C example:

```
   .
   .
   .
/* Specify CHARACTER LENGTH CHARACTERS in the DECLARE MODULE statement.
In addition, specify the  IDENTIFIER, NATIONAL, and DEFAULT character sets.
*/
exec sql DECLARE MODULE CCC_COLOURS
         NAMES ARE DEC_KANJI
         NATIONAL CHARACTER SET KANJI
         SCHEMA RDB$SCHEMA
         AUTHORIZATION SQL_SAMPLE
         CHARACTER LENGTH CHARACTERS
         DEFAULT CHARACTER SET DEC_KANJI
         ALIAS RDB$DBHANDLE;

/* If you do not specify character sets in the DECLARE ALIAS statement, SQL
*  uses the character sets of the compile-time database.
*/
exec sql DECLARE ALIAS FILENAME mia_char_set;

/* When you declare a parameter with lowercase char, SQL considers the
*  character set unspecified and allocates single-octet characters.
*/
char  english_p[31];
```

```
/* When you specify the character set, SQL allocates single- or multi-octet
* characters, depending upon the character set.
*/
char  CHARACTER SET DEC_MCS    french_p[31];
char  CHARACTER SET KANJI      japanese_p[31];
char  CHARACTER SET DEC_KANJI  dec_kanji_p[31];
   .
   .
   .
```

4.4.2 Supported Ada Variable Declarations

SQL lets you declare host language variables directly or by calling the Ada package, SQL_STANDARD.

You must use the SQL_STANDARD package if you want to conform to the ANSI/ISO SQL standard. This package defines the data types that are supported by the ANSI/ISO SQL standard. To use the package, first copy the file SYS$COMMON:[SYSLIB]SQL$STANDARD.ADA to your own Ada library, and then compile the package.

The package SQL_STANDARD declares the following ANSI-standard data types:

- CHAR
- SMALLINT

  The data type SMALLINT contains one subtype: INDICATOR_TYPE.
- INT
- REAL
- DOUBLE_PRECISION
- SQLCODE_TYPE

  The data type SQLCODE_TYPE contains two subtypes: NOT_FOUND and SQL_ERROR.
- SQLSTATE_TYPE

If ANSI/ISO SQL compliance is not important for your application, you can declare host language variables directly. The following list describes the variable declaration syntax that the SQL precompiler supports in Ada:

- Standard package data types
  - STRING

- CHARACTER
- SHORT_SHORT_INTEGER
- SHORT_INTEGER
- INTEGER
- FLOAT

  By default, Ada recognizes the FLOAT data type as an F-floating representation of floating-point data. However, Ada also allows you to override the default and specify that FLOAT denotes an IEEE S-Floating representation by using the FLOAT_REPRESENTATION(IEEE_FLOAT) pragma or using ACS CREATE LIBRARY or SET PRAGMA commands. This default can also be overridden at installation time. SQL does not recognize whether or not you override the F-floating default for the FLOAT data type. If you do override the FLOAT default, you will get Ada compile-time errors. These compile-time errors can be overcome by using a /FLOAT=IEEE_FLOAT qualifier with the SQL$PRE command.

  To avoid problems with the ambiguity in the FLOAT data type, use the SYSTEM package F_FLOAT and IEEE_SINGLE_FLOAT data types.

- LONG_FLOAT

  By default, Ada recognizes the LONG_FLOAT data type as a G-floating representation of floating-point data. However, Ada also allows you to override the default and specify that LONG_FLOAT denotes an IEEE S-Floating representation by using the FLOAT_REPRESENTATION(IEEE_FLOAT) pragma or using ACS CREATE LIBRARY or SET PRAGMA commands. This default can also be overridden at installation time. In addition, if the FLOAT_REPRESENTATION is VAX_FLOAT (the default), Ada allows you to specify that the LONG_FLOAT data type be represented by a D-Floating format by specifying the LONG_FLOAT(D_FLOAT) pragma. SQL does not recognize whether or not you override the G-floating default for the LONG_FLOAT data type. If you do override the LONG_FLOAT default, you will get Ada compile-time errors. These compile-time errors can be overcome by using a /FLOAT qualifier with the SQL$PRE command to specify either D_FLOATING or IEEE_FLOATING as appropriate.

To avoid problems with the ambiguity in the LONG_FLOAT data type, use the SYSTEM package G_FLOAT, D_FLOAT, and IEEE_DOUBLE_FLOAT data types.

Note

SQL$PRE will issue a warning (%SQL-W-NOFLOAT) if you use a /FLOAT qualifier with an /ADA qualifier because the ADA command does not have a /FLOAT qualifier. But if you use a pragma FLOAT REPRESENTATION to override the default floating point formats you must use the /FLOAT qualifier to let SQL$PRE know about this floating point format since it does not recognize the pragma. Simply ignore the warning. In addition to supporting IEEE formats, SQL$PRE allows the default G_FLOAT format for 64-bit floating point types to be overridden using a combination of the pragma FLOAT REPRESENTATION specifying VAX_FLOAT and the pragma LONG FLOAT specifying D_FLOAT. To use this combination, specify an SQL$PRE qualifier of /FLOAT=D_FLOAT.

The following example shows an Ada program with embedded SQL that will work correctly with SQL$PRE/ADA/FLOAT=IEEE:

```
PRAGMA FLOAT REPRESENTATION IEEE_FLOAT;
WITH SYSTEM; USE SYSTEM;
WITH STANDARD; USE STANDARD;
WITH SQL_STANDARD; USE SQL_STANDARD;
 . . .
PROCEDURE TESTIT IS
EXEC SQL BEGIN DECLARE SECTION;
KEYFIELD   : STRING(1..10);
FLOATER    : LONG_FLOAT;          -- package STANDARD
SQLFLOATER : REAL;                -- package SQL_STANDARD
GFLOATER   : G_FLOAT;             -- package SYSTEM
SFLOATER   : IEEE_SINGLE_FLOAT;   -- package SYSTEM
TFLOATER   : IEEE_DOUBLE_FLOAT;   -- package SYSTEM
EXEC SQL END DECLARE SECTION;
 . . .
BEGIN
 . . .
KEYFIELD := "1.0       ";
EXEC SQL SELECT FLOAT1, FLOAT2 INTO :SQLFLOATER, :GFLOATER
    WHERE KEYFIELD = :KEYFIELD;
 . . .
KEYFIELD := "2.0       ";
EXEC SQL SELECT FLOAT1, FLOAT2 INTO :SFLOATER, :TFLOATER
    WHERE KEYFIELD = "KEYFIELD;
 . . .
```

```
KEYFIELD := "3.0       ";
EXEC SQL SELECT FLOAT1, FLOAT2 INTO :FLOATER, TFLOATER
    WHERE KEYFIELD = KEYFIELD;
```

- Date-time data types

  The precompiler translates lines in a precompiled program that contain any of the date-time data types.

  Note

  Oracle Rdb reserves the right to change the code generated in translation of date-time data types at any time, without prior notice.

  – SQL_DATE, SQL_DATE_ANSI, SQL_DATE_VMS
  – SQL_TIME, SQL_TIMESTAMP
  – SQL_INTERVAL (DAY TO SECOND)

    Use this data type for variables that represent the difference between two dates or times. (Table 4–2 lists all the supported INTERVAL data types.)

- SQL definition package

  The precompiler generates a package that includes definitions for the following data types if Ada object declarations refer to them:

  – SQL_VARCHAR_n

    Use this data type for variables that correspond to VARCHAR and LONG VARCHAR columns in a database, where *n* is the length specified in the definition of the columns (always 16,383 characters for LONG VARCHAR columns).

    SQL declares a two-field Ada record when it encounters SQL_VARCHAR_n, with one field, *t*, containing the character string, and the second field, *l*, containing an integer denoting the length of the string.

    You can refer to the *l* field to determine the actual length of a varying character string, and refer to the *t* field to refer to the string itself. This excerpt from the online sample program sql_all_datatypes.sqlada refers to the *l* field to see if the value in an SQL_VARCHAR_n field is null.

```
        .
        .
        .
-- Variables for main program use
    type ALL_DATATYPES_RECORD_TYPE IS
        record
        .
        .
        .
            VARCHAR_VAR : sql_varchar_40;
        end record;

        .
        .
        .
-- The following if statements evaluate the contents of main variables
-- and then set indicators as appropriate.
        .
        .
        .
    if all_datatypes_record.varchar_var.l = 0 then
        indicator_group(7) := -1; end if;
```

- SQLDA_ACCESS

  Specifying this data type declares an SQLDA structure. It offers an advantage over an embedded INCLUDE SQLDA statement because you can use it in more than one declaration to declare multiple SQLDA structures.

- CDD_TYPES package data types (must specify WITH CDD_TYPES)
  - DATE_TIME_DATATYPE (Oracle Rdb recommends that you use SQL_TIMESTAMP)
  - SHORT_INTEGER_ARRAY (for indicator arrays only)
- SYSTEM package data types (must specify WITH SYSTEM)
  - D_FLOAT
  - G_FLOAT
  - F_FLOAT
  - IEEE_SINGLE_FLOAT
  - IEEE_DOUBLE_FLOAT
- Arrays

  Single-dimension arrays are supported to declare an indicator array to refer to a structure in SQL statements. The elements of the array must be declared as word integers (SHORT_INTEGER).

Character arrays are supported as types or subtypes but cannot refer to derived types.

SQL does not allow references to unconstrained arrays.

- Types

  The precompiler recognizes types for all the preceding data types plus records, derived types, and arrays.

  - Records can refer to any recognized type.
  - Derived types (NEW keyword) can refer to any recognized type. SQL allows but ignores range constraints in derived types.

  SQL does not allow references to types that use discriminants in any way or to access types. SQL does not allow references to integer (RANGE keyword), floating-point (DIGITS keyword), or fixed-point (DELTA keyword) types.

- Subtypes

  Subtypes can refer to any recognized type. SQL allows but ignores range constraints in subtypes.

- Assignments from expressions in declarations
- Context structure types

  When you write applications for the Ada precompiler, you should declare a context structure by declaring a variable of data type SQLCONTEXT_REC instead of declaring a structure. When you declare a variable with the data type SQLCONTEXT_REC, the Ada precompiler generates a context structure for you. For example, you declare the variable using the following code:

```
context_struc.sqlcontext_ver := 1;
context_struc.sqlcontext_tid.sqlcontext_tid_type := 1
context_struc.sqlcontext_tid.sqlcontext_tid_len := 16;
context_struc.sqlcontext_tid.sqlcontext_tid_value(1) := 0;
context_struc.sqlcontext_tid.sqlcontext_tid_value(2) := 0;
context_struc.sqlcontext_tid.sqlcontext_tid_value(3) := 0;
context_struc.sqlcontext_tid.sqlcontext_tid_value(4) := 0;
context_struc.sqlcontext_end := 0;
```

The following example illustrates some Ada declarations to which the SQL precompiler lets SQL statements refer:

```
-- Record with STANDARD, SYSTEM, and SQL package data types:
    type ALL_DATATYPES_RECORD_TYPE IS
        record
            CHAR_VAR : string(1..10);
            SMALLINT_VAR : short_integer;
            INTEGER_VAR : integer;
            REAL_VAR : system.F_float;
            DOUBLE_PREC_VAR : system.G_float;
            BIN_DATE_VAR : sql_date;
            VARCHAR_VAR : sql_varchar_40;
         end record;

    ALL_DATATYPES_RECORD : all_datatypes_record_type;

-- Derived type (SQL ignores RANGE specification):
type my_int is new integer range 1..20000;

-- Record using derived type:
type p_type is
    record
    pnum : string(1..6);
    pname : string(1..20);
    color : string(1..6);
    weight : my_int;
    city : string(1..20);
    end record;
p : p_type;

-- Indicator structure for handling null values:
    type INDICATOR_GROUP_TYPE is array(1..7) of short_integer;
    INDICATOR_GROUP : indicator_group_type := (0,0,0,0,0,0,0);

-- Indicator arrays:
IND2 : array (1..5) of short_integer;
subtype SUB_SHORT_INT is short_integer range -1..2000;
type MY_SUB_IND_TYPE is array(1..5) of SUB_SHORT_INT;
IND4 : array(1..5) of SUB_SHORT_INT;
ind5 : MY_SUB_IND_TYPE;

-- Character array:
CHAR1 : array(1..20) of character;

-- Character array referring to subtype:
subtype X is character;
type CHAR2 is array(1..20) of X;
P : CHAR2;
```

Here are examples of invalid declaration syntax. The comment preceding each declaration notes the reason an SQL statement cannot refer to the variable specified by the declaration or to a variable dependent on the declaration.

```
-- Enumerated type:
type ENUM_TYPE is (T1, T2, T3);
ENUM_TYPE_OBJ : ENUM_TYPE;

-- Integer type (RANGE allowed only in derived type or subtype):
type INTEGER_TYPE is range 1..20;
INTEGER_TYPE_OBJ : INTEGER_TYPE;

-- Type with DIGITS:
type FLOAT_TYPE is digits 10;
FLOAT_TYPE_OBJ : FLOAT_TYPE;

-- Access type:
type ACCESS_TYPE is access integer;
ACCESS_TYPE_OBJ : ACCESS_TYPE;

-- Discriminants in a record declaration:
type DISCR(X, Y, Z : integer := (X + 20)/2; W : natural) is
    record
    R1 : array(1..W) of character;
    R2 : integer := X;
    end record;
DISCR_OBJ : DISCR;

-- Variant records
type VAR_REC is
    record
    I : integer;
    case I is
        when 1 =>
            X : integer;
        when (3*j) =>
            Y : integer;
        when others
            case J is
                when 1 =>
                    Z1 : integer;
                when 2 =>
                    Z2 : integer;
            end case
    end case
    end record;
VAR_REC_OBJ : VAR_REC;

-- Multiple dimensioned array:
type MULDIM_ARR is array(1..20, 1..20) of integer;
MULDIM_ARR_OBJ : MULDIM_ARR;

-- Unconstrained array:
type UNC_ARRAY is array( integer range <>) of integer;
UNC_ARRAY_OBJ : UNC_ARRAY;

-- Unrecognized declaration of array (not SHORT_INT):
IND4 : array(1..5) of short_short_int;
```

Table 4–3 gives examples of Ada variable declarations that SQL supports for each SQL data type.

Table 4–3 Ada Declarations for SQL Data Types

SQL Example	Ada Example
CHAR (10)	STR1 : SQL_STANDARD.CHAR(1..10);[1]
VARCHAR (80)	STR2 : SQLVARCHAR_80;[2]
LONG VARCHAR	STR3 : SQLVARCHAR_16383;[2]
TINYINT (2)	Not supported
TINYINT	NUM1 : SHORT_SHORT_INTEGER;
SMALLINT (2)	Not supported
SMALLINT	NUM1 : SQL_STANDARD.SMALLINT;[1]
INTEGER (2)	Not supported
INTEGER	NUM2 : SQL_STANDARD.INT;[1]
BIGINT (2)	Not supported
BIGINT	Not supported
FLOAT (6)	NUM4 : SQL_STANDARD.REAL;[1]
FLOAT (25)	NUM4 : SQL_STANDARD.DOUBLE_PRECISION;[1]
REAL	NUM5 : SQL_STANDARD.REAL;[1]
DOUBLE PRECISION	NUM6 : SQL_STANDARD.DOUBLE_PRECISION;[1]
DATE	DATENUM1 : SQL_DATE;
DATE ANSI	DATENUM2 : SQL_DATE_ANSI;
DATE VMS	DATENUM3 : SQL_DATE_VMS;
TIME	DATENUM4 : SQL_TIME(0);
TIMESTAMP	DATENUM5 : SQL_TIMESTAMP(2);
INTERVAL DAY TO HOUR	DATENUM6 : SQL_INTERVAL (DAY TO HOUR);[3]
LIST OF BYTE VARYING	STR4 : SQL_STANDARD.CHAR(1..8);[4]

[1]The source file must explicitly use the SQL$STANDARD.ADA package to specify these types.

[2]The SQL precompiler defines the $SQL_VARCHAR data type as part of the package it generates during processing.

[3]Table 4–2 lists all the supported INTERVAL data types.

[4]This example shows how to retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. To retrieve the values of individual elements of that list, use host language variables of data type CHAR or VARCHAR.

The online sample program sql_all_datatypes.sqlada provides examples of declaring variables and using them in SQL statements. The program also illustrates a variety of SQL data definition and data manipulation statements. After SQL is installed, you can print, type, or search the program to find sample code related to a variety of topics.

The following example shows the commands to precompile, link, and run the sample program sql_all_datatypes.sqlada:

```
$ ACS CREATE LIBRARY [.ADALIB]
$ ACS SET LIBRARY [.ADALIB]
$ SQLPRE == "$SQL$PRE"
$ SQLPRE
INPUT FILE> sql_all_datatypes.sqlada/ADA
$ ! This LINK command requires that the logical name
$ ! LNK$LIBRARY is defined as SYS$LIBRARY:SQL$USER.OLB.
$ ACS LINK sql_all_datatypes sql_sql_all_datatypes.obj
$ RUN sql_all_datatypes.exe
        .
        .
        .
This is the stored row

CHAR_COL: Begin  end
SMALLINT_COL: -32768
INTEGER_COL: -2147483648
REAL_COL:  0.12346
DOUBLE_PRECISION_COL:  0.12345678901234
DATE_COL: 30-OCT-1987 09:00:00.00
VARCHAR_COL: This string is 39 characters in length.
        .
        .
        .
This is the row after update

CHAR_COL: NULL
SMALLINT_COL:  32767
INTEGER_COL:  2147483647
REAL_COL:  0.12346
DOUBLE_PRECISION_COL:  0.12345678901234
DATE_COL: 10/30/87 09:00:00
VARCHAR_COL: This string is 39 characters in length.
```

Note

When using the Ada precompiler, you must have unique procedure names for each subprogram. If your program uses the same procedure names for various subprograms, you have several alternatives:

- Ideally, use the module language instead of precompiled Ada. This restriction does not apply when using module language. See Chapter 3 for more information on SQL module language.
- Alternatively, you can use the separate compile-time feature of Ada. This feature precompiles all subprograms separately. However, if you use this alternative, SQL statements in the subprogram will be unable to reference types, variables, and so forth declared in the main program unit because they will be unknown.
- Another alternative is to make sure that all names used in SQL statements are unique. If your application must conform to the ANSI standard, the names of all host language variables used in SQL statements must be unique in the file.

4.4.3 Supported C Variable Declarations

Note

C is a case-sensitive language. The names of objects declared in a C program are case sensitive, but the names of SQL tables and other names are not case sensitive. Therefore, you must be careful about C constructs that you specify in SQL statements. These constructs include variable names and labels of program sections. See the *Oracle Rdb Guide to SQL Programming* for more information about declaring C variables.

The following list describes the variable declaration syntax for character data types that the SQL precompiler supports in C:

- char x[n]
- char *x, assumes LONG VARCHAR type (that is, char x[16...])
- char CHARACTER SET character-set-name clause

  The CHARACTER SET character-set-name clause is optional.
- $SQL_VARCHAR (n)
- $SQL_VARCHAR (n) CHARACTER SET character-set-name

The CHARACTER SET clause is optional.

For information about the supported character sets, see Section 2.1.

The following list describes the variable declaration syntax that the SQL precompiler supports in C:

- Data type keywords (see Table 4–4)
- Storage class identifiers and modifiers
- struct
- union
- type def
- Initial value assignments
- Arrays

  Only single-dimension arrays are supported and only to declare an indicator array for use with a reference to a structure in SQL statements. Furthermore, the size of the array must be specified explicitly. Although you can use any data type for indicator array elements, Oracle Rdb recommends that you use variables of the INTEGER data type.
- Pointers

  Only a single level of pointer variables are supported and only those that point to elementary data types.

  Because C pointer variables cannot specify length attributes, SQL sometimes must allocate the largest possible piece of memory to process statements that refer to char pointer variables. SQL cannot determine the length of char pointer variables and allocates 16,383 bytes of memory for each variable in the following cases:

  – The SQL statement contains a concatenated value expression or a substring.

  – The SQL statement refers to the char pointer variable in a predicate, such as `WHERE EMP_ID = :POINTER_VAR`.

  – The SQL statement converts the contents of the char pointer variable to a numeric data type in the database.

  Avoid the use of char pointer variables in these cases because such a large memory allocation for char pointer variables wastes memory and degrades performance, especially for remote database access.

- Valid declaration syntax

  The following are examples of valid declaration syntax:

```
a_var[10];

$SQL_VARCHAR(10) x,y,z;

int            SQLCODE;

struct
     {
     char       b_var[5];
     short int  c_var;
     } a_record;

union
     {
     char         string_date[17];
     struct
          {
          char     year_var1[2];
          char     year_var2[2];
          char     month_var[2];
          char     day_var[2];
          char     hour_var[2];
          char     minute_var[2];
          char     second_var[2];
          char     hundredth_var[2];
          } date_group;
      } date_union;

int               indicator_item[2];

globaldef double  c_var;

static            d_var;

char              *x;
```

- Invalid declaration syntax

Table 4–4 Supported C Datatypes

C type or typedef	SQL type	Comments and Restrictions
char	CHARACTER	
char *	LONG VARCHAR	Assumed to be VARCHAR (16383)

(continued on next page)

Table 4–4 (Cont.) Supported C Datatypes

C type or typedef	SQL type	Comments and Restrictions
char [n]	CHARACTER	*n* must be be an integer literal; #define names or expressions are not supported.
int	INTEGER	Cannot be specified as unsigned.
short	SMALLINT	Cannot be specified as unsigned.
short int	INTEGER	Cannot be specified as unsigned.
long int	INTEGER	Cannot be specified as unsigned.
float	REAL	
double	DOUBLE PRECISION	
enum	INTEGER	
long	INTEGER or BIGINT	On OpenVMS the data type **long** is 32 bits
int8	TINYINT	Requires #include <ints.h>
int16	SMALLINT	Requires #include <ints.h>
__int16	SMALLINT	
int32	INTEGER	Requires #include <ints.h>
__int32	INTEGER	
int64	BIGINT	OpenVMS Alpha platform only. Requires #include <ints.h>
__int64	BIGINT	
	$SQL_VARCHAR (n)	The CHARACTER SET clause is optional.
	SQL_DATE SQL_DATE_ANSI SQL_DATE_VMS SQL_TIME SQL_TIMESTAMP	The SQL precompiler will transform the pseudo types in natvie C datatypes

(continued on next page)

Table 4–4 (Cont.) Supported C Datatypes

C type or typedef	SQL type	Comments and Restrictions
	SQL_INTERVAL (DAY TO SECOND)	Use this data type for variables that represent the difference between two dates or times. Table 4–2 lists all the supported INTERVAL data types.

The precompiler accepts but ignores some syntax that it does not support if the syntax is unimportant to SQL operations. For example, the precompiler does not consider implementation of storage class modifiers important to SQL operations. The precompiler accepts such modifiers in declarations but ignores them. In contrast, implementation of data type syntax must be understood by SQL for SQL to use the variable correctly. Therefore, lack of SQL support for the enumerated data type means that the precompiler considers a declaration invalid if it contains the keyword enum.

For all invalid declarations, the precompiler does not return an error following the declarations themselves, but rather following the SQL statements that refer to the declarations.

The precompiler does not recognize the #define directive. For example, defining `{` and `}` as BEGIN and END is not supported.

The following are examples of invalid declaration syntax. The comment following each declaration notes the reason an SQL statement cannot refer to the variable specified by the declaration or to a variable dependent on the declaration.

```
int           indicator_item[];    /* implicit dimension for array */
char          func_ret();          /* function return status */
int           mult_arr[5][10];     /* multidimensional array */
unsigned int  uns_var;             /* unsigned data */
int           bit3 : 4, bit4 : 3;  /* bit fields */
enum          {a, b, c} enum_var;  /* enumerated data type */
char          *a_prt[5];           /* array of pointers */
int           **x;                 /* two levels of pointers */
struct        x_rec *x             /* pointer to structures */
foo (char *x, int y)               /* declarations within functions */
```

Table 4–5 gives examples of C variable declarations that SQL supports for each SQL data type.

Table 4–5 C Declarations for SQL Data Types

SQL Example	C Example
CHAR (10)	char str1[11] SQL expects character strings to be in ASCIZ format. You therefore declare a char host language variable for a CHAR column to be one character more than the column size. (This allows space for the null character that terminates ASCIZ strings.) You can avoid this restriction when you copy definitions from the data dictionary by specifying the FIXED argument in your SQL INCLUDE statement, if you prefer. The character set is UNSPECIFIED.
CHAR (10) CHARACTER SET KANJI	char CHARACTER SET KANJI str[11][1] This data type has the same characteristics as char, except that the character set is that specified in the CHARACTER SET clause.
VARCHAR(10)	$SQL_VARCHAR(10) When you use the typedef $SQL_VARCHAR(max_length_of_varchar), the SQL precompiler declares a macro at the beginning of the file defining a host language variable that is a word length, called len, followed by a number of bytes of data, called data. In this example, 10 bytes of data follow the len. This is the only C variable declaration supported by SQL that is appropriate for storing and passing binary data. The character set is UNSPECIFIED.

[1] See Section 4.4.1 for information about character length and the precompiler.

(continued on next page)

Table 4–5 (Cont.) C Declarations for SQL Data Types

SQL Example	C Example
VARCHAR(10) CHARACTER SET KANJI	$SQL_VARCHAR(10) CHARACTER SET KANJI str[1] This data type has the same characteristics as $SQL_VARCHAR, except that the character set is that specified in the CHARACTER SET clause.
LONG VARCHAR	char str3[16384] Because SQL expects character strings to be in ASCIZ format, it uses the null string terminator to determine the length of the value stored in LONG VARCHAR columns. You therefore declare a host language variable for the LONG VARCHAR data type as a fixed-length char variable. The variable should be large enough to contain the largest valid string allowed in the column, plus one character for the null terminator. The character set is UNSPECIFIED.
TINYINT	char num1 SQL supports scale factors on TINYINT columns, but C does not.
SMALLINT	short num1 SQL supports scale factors on SMALLINT columns, but C does not. If the SMALLINT column is scaled, declare the variable as float rather than short.
INTEGER	long num2 SQL supports scale factors on INTEGER columns, but C does not. If the INTEGER column is scaled, declare the variable as double rather than int.

[1] See Section 4.4.1 for information about character length and the precompiler.

(continued on next page)

Table 4–5 (Cont.) C Declarations for SQL Data Types

SQL Example	C Example
BIGINT	double num3 C does not support the BIGINT data type. The C data type double is appropriate for both scaled and unscaled BIGINT columns, and SQL performs the conversion for you. However, the double data type provides only approximations of larger values that can be stored in a BIGINT column.
FLOAT (6) FLOAT (25)	float num4 double num4
REAL	float num5
DOUBLE PRECISION	double num6
DATE	SQL_DATE datenum1
DATE ANSI	SQL_DATE_ANSI datenum2
DATE VMS	SQL_DATE_VMS datenum3
TIME	SQL_TIME datenum4
TIMESTAMP	SQL_TIMESTAMP datenum5
INTERVAL DAY TO HOUR	SQL_INTERVAL (DAY TO HOUR) datenum6 Table 4–2 lists all of the supported INTERVAL data types.
LIST OF BYTE VARYING	char datenum[8] C does not support the LIST OF BYTE VARYING data type. Declaring an 8-byte character variable for a LIST OF BYTE VARYING column gives SQL sufficient space to store the segmented string identifier that points to the first element of the list. You can use host language variables of data type CHAR or VARCHAR to retrieve the values of individual elements of that list.

The online sample program sql_all_datatypes.sc provides examples of declaring variables and using them in SQL statements. The program also illustrates a variety of SQL data definition and data manipulation statements. After SQL is installed, you can print, type, or search the program to find sample code related to a variety of topics.

The following restrictions apply to C variables:

- When you use the SQL precompiler for C and specify a C module language, SQL usually translates C character strings as null-terminated strings. This means that when SQL passes these character strings from the database to the program, it reserves space at the end of the string for the null character. When a program passes a character string to the database for input, SQL looks for the null character to determine how many characters to store in the database. SQL stores only those characters that precede the null character; it does not store the null character.

  The only exception to this restriction is when you copy data definitions from the data dictionary. The SQL INCLUDE statement gives you the option of changing the default translation of character data to fixed-character format, if you prefer. For more information, see the FIXED and NULL TERMINATED BYTES arguments in the INCLUDE Statement.

  If you use SQL module language instead of the SQL precompiler, you can also specify that the length field be interpreted as a character count. For more information, see the NULL TERMINATED CHARACTERS argument in Section 3.2.

  Because of the way SQL translates C character strings, you may encounter problems with applications that pass binary data to and from the database. To avoid these problems when you use the SQL precompiler for C, use the $SQL_VARCHAR data type that SQL provides.

  The SQL INCLUDE statement AS name clause allows you to rename the dictionary record that you retrieve if you do not want the structure name to be the same as the dictionary record name. For more information, see the INCLUDE Statement.

- The SQL precompiler for the C language gives the following error message when an SQL statement refers to a host language variable declared as a character array whose declaration includes anything other than a straight numeric value:

  ```
  %SQL-F-BAD_ARRAY, Host variable address contains an array syntax error
  in its declaration.
  ```

  For example, this error occurs when the declaration contains a named constant or an expression:

```
#define NAME_LEN     (20)
#define ADDRESS_LEN  (31)
      char name [NAME_LEN + 1]          /* This cannot be used */
      char address [ADDRESS_LEN]        /* This cannot be used */
```

There is a solution that requires two actions:

1. Remove the expressions from the declarations and update the #define line accordingly; also remove the parentheses from the #define line:

   ```
   #define NAME_LEN      21
   #define ADDRESS_LEN   31
         char name [NAME_LEN]
         char address [ADDRESS_LEN]
   ```

2. Run the C code through the C preprocessor before invoking the SQL precompiler. This forces all named constants to be translated before the precompiler tries to use them:

   ```
   CC/PREPROCESS=filename.SCP filename.SC
   SQL$PRE/CC filename.SCP
   ```

4.4.4 Supported COBOL Variable Declarations

The following list describes the variable declaration syntax for character data types that the SQL precompiler supports in COBOL:

- PICTURE IS can be abbreviated as PICTURE or PIC.
- CHARACTER SET character-set-name PICTURE IS.
- PICTURE clauses for numeric variables must begin with S (must be signed) and cannot include P characters.
- PICTURE clauses cannot include editing characters.

For information about the supported character sets, see Section 2.1.

The following list describes the variable declaration syntax that the SQL precompiler supports in COBOL:

- PICTURE IS clause
  - PICTURE IS can be abbreviated as PICTURE or PIC.
  - PICTURE clauses for numeric variables must begin with S (must be signed) and cannot include P characters.
  - PICTURE clauses cannot include editing characters.

- USAGE IS clause
  - USAGE IS must immediately follow a PICTURE clause.
  - USAGE IS can be abbreviated as USAGE or omitted completely.
  - USAGE IS must have as an argument BINARY, COMPUTATIONAL, COMPUTATIONAL-1, COMPUTATIONAL-2, or COMPUTATIONAL-3. COMPUTATIONAL can be abbreviated as COMP in all USAGE IS or DISPLAY declarations. BINARY is a synonym for COMPUTATIONAL or COMP.
- VALUE IS clause

  VALUE IS can be abbreviated as VALUE and is allowed without restriction.
- IS EXTERNAL clause

  IS EXTERNAL can be abbreviated as EXTERNAL and is allowed without restriction.
- IS GLOBAL clause

  IS GLOBAL can be abbreviated as GLOBAL and is allowed without restriction.
- SIGN clause

  SIGN is allowed but must immediately follow a PICTURE clause or a USAGE IS clause.
- Group data items
  - Group data items are allowed without restriction.
  - Variables associated with the SQL VARCHAR and LONG VARCHAR data types must be declared as group data items with two elementary items at level 49. The first elementary item must be a small integer to contain the actual length of the character string. The second elementary item must be a character string long enough to contain the string itself.

```
* Declaration for an SQL column
* defined as VARCHAR (80):
*
01 VARYING_STRING.

        49 STRING_LENGTH   PIC S9(4) USAGE IS COMP.
        49 STRING_TEXT     PIC X(80).
```

- OCCURS n TIMES clause
  - OCCURS clauses are permitted only for declarations of indicator arrays. Although you can use any data type for indicator array elements, Oracle Rdb recommends that you declare them as integers (PIC S9(9) COMP).
  - Multidimension tables (nested OCCURS clauses) and variable-occurrence data items (OCCURS DEPENDING ON clause) are not supported.
- REDEFINES clauses

  You can refer to host language variables that have a REDEFINES clause or that are subordinate to a REDEFINES clause.
- SQL date-time data types
  - SQL_DATE, SQL_DATE_ANSI, SQL_DATE_VMS
  - SQL_TIME, SQL_TIMESTAMP
  - SQL_INTERVAL (DAY TO SECOND)

    Use this data type for variables that represent the difference between two dates or times. (Table 4–2 lists all the supported INTERVAL data types.)

  The precompiler replaces these data types with host language data declarations that are supported in the compilers themselves.

The following example illustrates some COBOL declarations that SQL will and will not accept:

```
* SQL will accept:
01 A PIC S9(7)V99 COMP.
* SQL will not accept (unsigned numeric):
01 B PIC 9(7)V99 COMP.
*
*
* SQL will accept:
01 C COMP-1 VALUE IS -1.
* SQL will not accept (implicit USAGE IS DISPLAY):
01 E PIC S9(4).
*
*
* SQL will accept:
01 indicators-x.
   05 indicator-null   pic s9(9) comp occurs 40.
* SQL will not accept
01 indicators-x.
   05 indicator-null   pic s9(9) comp occurs 40 indexed by x1.
```

```
*
* SQL will accept (host structure and indicator array):
01      F       EXTERNAL.
        02      F1      PIC S9(9) COMP.
        02      F2      PIC X(20).
        02      F2_R    REDEFINES F2.
                03      G1      PIC X(10).
                03      G2      COMP-2.
        02      F3      PIC S9(9) COMP.
01 F_IND_ARRAY
        02 F_IND OCCURS 3 TIMES PIC S9(9) COMP.
```

Table 4–6 gives examples of COBOL variable declarations that SQL supports for each SQL data type.

Table 4–6 COBOL Declarations for SQL Data Types

SQL Example	COBOL Example
CHAR (10)	01 STR1 PICTURE X(10). The character set is UNSPECIFIED.
CHAR (10) CHARACTER SET KANJI	01 STR1 CHARACTER SET KANJI PICTURE X(10).[5]
VARCHAR (80)	01 STR2. 49 STR2L PICTURE S9(4) COMP. 49 STR2C PICTURE X(80). The character set is UNSPECIFIED.
VARCHAR (80) CHARACTER SET KANJI	01 STR2. 49 STR2L PICTURE S9(4) COMP. 49 STR2C CHARACTER SET KANJI PICTURE X(80).[5]
LONG VARCHAR	01 STR3. 49 STR3L PICTURE S9(4) COMP. 49 STR3C PICTURE X(16383). The character set is UNSPECIFIED.
TINYINT (2)	Not supported by COBOL.
TINYINT	Not supported by COBOL.
SMALLINT (2)	01 NUM1 PICTURE S99V99 COMP.
SMALLINT	01 NUM1 PICTURE S9(4) COMP.

[5]See Section 4.4.1 for information about character length and the precompiler.

(continued on next page)

Table 4–6 (Cont.) COBOL Declarations for SQL Data Types

SQL Example	COBOL Example
INTEGER (2) INTEGER	01 NUM2 PICTURE S9(7)V99 COMP. 01 NUM2 PICTURE S9(9) COMP.
BIGINT (2) BIGINT	01 NUM3 PIC S9(16)V99 COMP. 01 NUM3 PIC S9(18) COMP.
FLOAT (6) FLOAT (25)	01 NUM4 COMP-1. 01 NUM4 COMP-2.[1]
REAL	01 NUM5 COMP-1.
DOUBLE PRECISION	01 NUM6 COMP-2.[2]
DATE	01 DATENUM1 SQL_DATE.
DATE ANSI	01 DATENUM2 SQL_DATE_ANSI.
DATE VMS	01 DATENUM3 SQL_DATE_VMS.
TIME	01 DATENUM4 SQL_TIME(0).
TIMESTAMP	01 DATENUM5 SQL_TIMESTAMP(2).
INTERVAL DAY TO HOUR	01 DATENUM6 SQL_INTERVAL (DAY TO HOUR).[3]
LIST OF BYTE VARYING	01 STR4 PICTURE X(8).[4]

[1]COMP-2 is FLOAT (25).

[2]COMP-2 is DOUBLE PRECISION.

[3]Table 4–2 lists all the supported INTERVAL data types.

[4]This example shows how to retrieve the segmented string (list) identifier, a pointer to the first element of the list, using an 8-byte character string. (You could use a BIGINT instead if you prefer.) To retrieve the values of individual elements of that list, use host language variables of data type CHAR or VARCHAR.

The online sample program sql_all_datatypes.sco provides examples of declaring variables and using them in SQL statements. The program also illustrates a variety of SQL data definition and data manipulation statements. After SQL is installed, you can print, type, or search the program to find sample code related to a variety of topics.

In COBOL, a structure defined as word followed by a string is treated as a single variable. The type equates to VARCHAR(n).

4.4.5 Supported FORTRAN Variable Declarations

The following list describes the variable declaration syntax for character data types that the SQL precompiler supports in FORTRAN:

- CHARACTER
- CHARACTER character-set-name

For information about the supported character sets, see Section 2.1.

The following list describes the variable declaration syntax that the SQL precompiler supports in FORTRAN:

- Declarations – See Table 4–7.
- Initial values assigned in the declaration
- STRUCTURE declarations
- UNION declarations within structures
- RECORD statements
- DIMENSION statements
  - DIMENSION statements are permitted only for declarations of indicator arrays. Although you can use any data type for indicator array elements, Oracle Rdb recommends that you use variables of the INTEGER data type.
  - Multidimension arrays and dynamic-sized arrays are not supported.

Table 4–7 Supported FORTRAN Datatypes

FORTRAN type	SQL type	Comments and Restrictions
BYTE	TINYINT	
CHARACTER*n	CHAR	The **n** represents a positive integer literal
INTEGER	INTEGER	
INTEGER*1	TINYINT	
INTEGER*2	SMALLINT	

(continued on next page)

Table 4–7 (Cont.) Supported FORTRAN Datatypes

FORTRAN type	SQL type	Comments and Restrictions
INTEGER*4	INTEGER	
INTEGER*8	BIGINT	OpenVMS Alpha only
LOGICAL	INTEGER	
LOGICAL*1	TINYINT	
LOGICAL*2	SMALLINT	
LOGICAL*4	INTEGER	
LOGICAL*8	BIGINT	OpenVMS Alpha only
REAL	REAL	
REAL*4	REAL	
REAL*8	DOUBLE PRECISION	
STRUCTURE /name/ integer*2 len character*n body END STRUCTURE	VARCHAR	The named structure can be used to define other FORTRAN host variables. The **len** component of the structure must be set to the correct length of the string before use as a parameter to SQL. The **n** represents a positive integer literal
	SQL_DATE SQL_DATE_ANSI SQL_DATE_VMS SQL_TIME SQL_TIMESTAMP	The SQL precompiler will transform the pseudo types in native FORTRAN datatypes.
	SQL_INTERVAL (DAY TO SECOND)	Use this data type for variables that represent the difference between two dates or times. Table 4–2 lists all the supported INTERVAL data types.

Implicit declarations are not supported. SQL generates a “host variable was not declared” error when it encounters an implicitly declared variable in an SQL statement.

The following example illustrates some FORTRAN declarations that SQL will and will not accept:

```
C SQL will accept:
CHARACTER*1 F1, F2*2 /'XX'/, F3*2345
LOGICAL B1*2/1/
C
C SQL will accept:
REAL D1*4, D2*8
C
C SQL will not accept (REAL declaration bigger than 8 bytes):
REAL D3*16
C SQL will not accept (COMPLEX data type unsupported):
COMPLEX E1*8, E2*16, E3, E4*8(16)
C
C
C SQL will accept:
C host structure:
STRUCTURE /M1_STRUCT/
    STRUCTURE /M11_STRUCT/ M11
        CHARACTER       M111*6, M112*20, M113*6
    END STRUCTURE
    INTEGER             M12*2
    UNION
    MAP
    CHARACTER           M13*15
    END MAP
    MAP
    INTEGER             M13_A*4
    END MAP
    END UNION
END STRUCTURE
C records based on structures:
RECORD /M1_ STRUCT/ M_1, M_2
RECORD /M11_STRUCT/ M_4
C
C
C SQL will accept (indicator array):
INTEGER*4 L1, L2
DIMENSION L1(10), L2(-3:7)
C
C SQL will not accept (dynamic-sized array):
CHARACTER F4*(*)
C SQL will not accept (multidimension array):
INTEGER*2 L5
DIMENSION L5(2,5)
C SQL will not accept (arrays of structures):
RECORD /M1_STRUCT/ M_2(10)
```

Table 4–8 gives examples of FORTRAN variable declarations that SQL supports for each SQL data type.

Table 4–8 FORTRAN Declarations for SQL Data Types

SQL Example	FORTRAN Example
CHAR (10)	CHARACTER*10 STR1 The character set is UNSPECIFIED.
CHAR (10) CHARACTER SET KANJI	CHARACTER*10 CHARACTER SET KANJI STR1[6]
VARCHAR (10)	CHARACTER*10 STR2 The character set is UNSPECIFIED.
VARCHAR (10) CHARACTER SET KANJI	CHARACTER*10 CHARACTER SET KANJI STR2[6]
LONG VARCHAR	CHARACTER*16383 STR3 The character set is UNSPECIFIED.
TINYINT	LOGICAL*1 NUM1[1]
SMALLINT	INTEGER*2 NUM1[2]
INTEGER	INTEGER*4 NUM2[2]
BIGINT	INTEGER*8 NUM3
FLOAT (6) FLOAT (25)	REAL*4 NUM4 REAL*8 NUM4 or DOUBLE PRECISION NUM4
REAL	REAL*4 NUM5
DOUBLE PRECISION	DOUBLE PRECISION NUM6
DATE	SQL_DATE DATENUM1
DATE ANSI	SQL_DATE_ANSI DATENUM2
DATE VMS	SQL_DATE_VMS DATENUM3
TIME	SQL_TIME DATENUM4
TIMESTAMP	SQL_TIMESTAMP(2) DATENUM5
INTERVAL DAY TO HOUR	SQL_INTERVAL (DAY TO HOUR) DATENUM6[3]

[1]In FORTRAN, data type BYTE is a synonym for LOGICAL*1 and is parsed by SQL.
[2]FORTRAN does not support scale factors on integer data types.
[6]See Section 4.4.1 for information about character length and the precompiler.

(continued on next page)

Table 4–8 (Cont.) FORTRAN Declarations for SQL Data Types

SQL Example	FORTRAN Example
LIST OF BYTE VARYING[3]	CHARACTER*8 STR4[4]

[4]Table 4–2 lists all the supported INTERVAL data types.

The online sample program sql_all_datatypes.sfo provides examples of declaring variables and using them in SQL statements. The program also illustrates a variety of SQL data definition and data manipulation statements. After SQL is installed, you can print, type, or search the program to find sample code related to a variety of topics.

Note

When using FORTRAN with the SQL precompiler, keep in mind that the FORTRAN compiler lets you specify a maximum number of continuation lines (up to 99) in a statement if you use the CONTINUATIONS qualifier. The default number of continuation lines is 19.

If a program uses a record definition, the SQL precompiler separates the record into individual elements and places each one on a separate line. If the number of elements in the record is greater than the maximum number of continuation lines, the FORTRAN compiler will generate an error.

If this happens, increase the number of continuation lines using the CONTINUATIONS qualifier to the FORTRAN command line. If the record contains more elements than the maximum allowed by FORTRAN (99 elements), you can edit the intermediate file (the .for file extension) to place more than one element on a line.

In FORTRAN, a structure defined as word followed by a string is treated as a single variable. The type equates to VARCHAR(n). For example, the structure in the following example is treated as a single variable:

```
STRUCTURE /struct_name
        INTEGER*2           length
        CHARACTER*32        string
END STRUCTURE
```

4.4.6 Supported Pascal Variable Declarations

The following list describes the variable declaration syntax that the SQL precompiler supports in Pascal:

- Data type keywords

  Declarations can include only the following Pascal data types:

  - INTEGER8, INTEGER16, INTEGER32, and INTEGER64
  - REAL
  - SINGLE
  - DOUBLE
  - F_FLOAT
  - D_FLOAT
  - G_FLOAT
  - S_FLOAT
  - T_FLOAT
  - CHAR
  - PACKED ARRAY [1..n] OF CHAR;
  - VARYING [u] OF CHAR
  - [BYTE] –128..127;
  - [WORD] –32768..32767;
  - Date-time data types (Table 4–2 lists these data types.)

  In addition, the SQL Pascal precompiler provides the following data types:

  - SQL_LONG_VARCHAR
  - SQL_DATE
  - SQL_SMALLINT
  - SQL_INDICATOR
  - SQL_BIGINT
  - SQL_QUAD
  - SQL_DATE, SQL_DATE_ANSI, SQL_DATE_VMS

– SQL_TIME, SQL_TIMESTAMP

– SQL_INTERVAL (DAY TO SECOND)

  Use this data type for variables that represent the difference between two dates or times. (Table 4–2 lists all the supported INTERVAL data types.)

- Records

  The SQL precompiler supports Pascal record definitions. It also supports nested records such as the following:

```
type_record_type = record
    employee_id :   employee_id_str;
    last_name   :   last_name_str;
    first_name  :   first_name_str;
    middle_init :   middle_init_str;
    address_dat1:   address_str;
    address_dat2:   address_str;
    city        :   city_str;
    state       :   state_str;
    postal_code :   postal_code_str;
    sex         :   sex_str;
    status_code :   status_code_str;
end;

name_rec = record
    last_name   :   last_name_str;
    first_name  :   first_name_str;
    middle_init :   middle_init_str;
end;

address_rec = record
    address_dat1    :   address_str;
    address_dat2    :   address_str;
    city            :   city_str;
    state           :   state_str;
    postal_code     :   postal_code_str;
end;

rec_in_rec = record
    employee_id :   employee_id_str;
    emp_name    :   name_rec;
    emp_addr    :   address_rec;
    sex         :   sex_str;
    status_code :   status_code_str;
end;

rec_in_rec_in_rec = record
    nested_again :   rec_in_rec;
end;
```

A record that is used in an SQL statement cannot contain a pointer to another record.

The SQL precompiler does not support variant records.

- Initial value assignments

  The SQL precompiler supports initial values assigned in the declaration:

  ```
  dateind : SQL_INDICATOR:=0;
  ```

- Arrays

  Packed arrays are supported to declare SQL character strings.

  Single-dimension arrays are supported to declare an indicator array to refer to a structure in SQL statements. The elements of the array must be declared as word integers [WORD]–32768..32767 or SQL_INDICATOR.

- Pointers

  The SQL precompiler for Pascal supports one level of pointers.

  ```
  type
       a = ^integer;

  var
       b :  a;  (* the use of the variable b is supported  *)
       c : ^a;  (* do not use any form of variable c in an SQL statement)
  ```

The following examples illustrate valid Pascal declaration syntax:

```
var
    pas_date : SQL_TIMESTAMP;
    pas_float: real;
    pas_flt  : single;
    pas_gflo : double;
    pas_int  : integer;
    pas_qword: SQL_BIGINT;
    pas_text : packed array [1..31] of char;
    pas_vtxt : varying [255] of char;
    pas_smal : [word] -32768..32767;

    dateind :  SQL_INDICATOR:=0;
    floaind :  [word] -32768..32767;
    gfloind :  SQL_INDICATOR;
    intind  :  SQL_INDICATOR;
    qind,txtind,vtxtind,smalind : SQL_INDICATOR;
```

Here are examples of invalid declaration syntax for Pascal:

```
type
   x     = ^my_rec;  (*forward declarations are not supported*)
   myrec = record
         a: integer;
         b: integer;
   end;
```

A record cannot point to itself. For example, the following declaration is not supported:

```
foo = record
  a : integer;
  b : SQL_SMALLINT;
  c : ^foo;
end;

bar = record
   a : integer;
   b : integer
   c : bar;
```

The SQL precompiler does not support the following:

- Attributes other than [HIDDEN]
- Ranges

The following example shows SQL statements embedded in Pascal host language statements. For readability, all Pascal host language statements are written in lowercase letters, and all SQL data types and embedded SQL statements are written in uppercase letters.

```
label
        leap_frog;

begin


dmp_alldtps;
insert_some_data;
dmp_alldtps;
EXEC SQL OPEN DTPS;
EXEC SQL FETCH DTPS INTO
        :PAS_DATE :DATEIND,
        :PAS_FLT:FLOAIND,
        :PAS_GFLO INDICATOR :GFLOIND,
        :PAS_INT:INTIND,
        :PAS_QWORD :QIND,
        :PAS_TEXT INDICATOR :TXTIND,
        :PAS_VTXT :VTXTIND,
        :PAS_SMAL INDICATOR :SMALIND ;
```

```
writeln('single  ',pasflt);
EXEC SQL CLOSE DTPS;

(*Note that an SQL statement can reside on the same line as a label*)
leap_frog: EXEC SQL ROLLBACK;

end.
```

All SQL statements embedded in a Pascal host language program must end with a semicolon (;). This means that if you want to place an SQL statement before an else action, you must surround it with a begin-end block:

```
if budget_actual < budget_total
then
    begin
    EXEC SQL INSERT ...;
    end
else
```

The online sample program sql_all_datatypes.spa provides examples of declaring variables and using them in SQL statements. The program also illustrates a variety of SQL data definition and data manipulation statements. After SQL is installed, you can print, type, or search the program to find sample code related to a variety of topics.

The following example shows the commands to precompile, link, and run the sample program sql_all_datatypes.spa:

```
$ SQLPRE :== $SQL$PRE
$ SQLPRE sql_all_datatypes/PASCAL
$ LINK sql_all_datatypes
$ RUN sql_all_datatypes
.
.
.
  Setting up the database.
  Declaring the schema.
  Declaring the cursor.
  Inserting data into table.

These are the stored rows:
-
```

```
date_ind =          0
pas_date     = 12-OCT-1988
pas_float    =  9.69600E+01
pas_gfloat   =  1.000000000000E+008
pas_int      = 2147483647
pas_qword l0 =    5000000
pas_qword l1 =          0
pas_text     = text in a packed array
pas_vartxt   = varying text
pas_small    =         19

date_ind =          0
pas_date     = 12-OCT-1988
pas_float    =  9.69600E+01
pas_gfloat   =  1.000000000000E+008
pas_int      = 2147483647
pas_qword    = null
pas_text     = Changed the text
pas_vartxt   = varying text
pas_small    =         19
single   9.69600E+01
```

Table 4–9 gives examples of Pascal variable declarations that SQL supports for each SQL data type.

Table 4–9 Pascal Declarations for SQL Data Types

SQL Example	Pascal Example
LONG VARCHAR	SQL_LONG_VARCHAR;[1]
TINYINT	SQL_TINYINT;[2]
SMALLINT	SQL_SMALLINT;[3]
SMALLINT	SQL_INDICATOR;[4]
SMALLINT	[WORD] –32768..32767;
BIGINT	SQL_BIGINT;[5]
INTEGER	INTEGER;
REAL	REAL;

[1]SQL_LONG_VARCHAR expands to [HIDDEN] VARYING [16383] OF CHAR;

[2]SQL_TINYINT expands to [HIDDEN, BYTE] –128..127;

[3]SQL_SMALLINT expands to [HIDDEN, WORD] –32678..32767;

[4]SQL_INDICATOR expands to [HIDDEN, WORD] –32678..32767;

[5]SQL_BIGINT is the only way to specify a BIGINT. SQL_BIGINT expands to [HIDDEN, QUAD, UNSAFE] RECORD L0:INTEGER;L1:INTEGER END; the user then can refer to the pieces by variable.L0 and variable.L1.

(continued on next page)

Table 4–9 (Cont.) Pascal Declarations for SQL Data Types

SQL Example	Pascal Example
REAL	SINGLE;
REAL	F_FLOAT;
REAL	S_FLOAT;
DOUBLE PRECISION	DOUBLE;
DOUBLE PRECISION	D_FLOAT;
DOUBLE PRECISION	G_FLOAT;
DOUBLE PRECISION	T_FLOAT;
CHAR/CHAR(1)	CHAR;
CHARACTER(n)	PACKED ARRAY [1..n] OF CHAR;
VARCHAR(u)	VARYING [u] OF CHAR;
DATE	SQL_DATE;
DATE ANSI	SQL_DATE_ANSI;
DATE VMS	SQL_DATE_VMS;
TIME	SQL_TIME(0);
TIMESTAMP	SQL_TIMESTAMP(2);
INTERVAL YEAR TO MONTH	SQL_INTERVAL (YEAR TO MONTH);[6]
INTERVAL DAY TO HOUR	SQL_INTERVAL (DAY TO HOUR);[6]
LIST OF BYTE VARYING	PACKED ARRAY [1..8] OF CHAR[7]

[6]Table 4–2 lists all the supported INTERVAL data types.

[7]Pascal does not support the LIST OF BYTE VARYING data type. This example shows how to retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. (If you prefer, you can use a BIGINT.) To retrieve the values of individual elements of that list, use host language variables of data type CHAR or VARCHAR.

Note

The Pascal precompiler for SQL gives an incorrect %SQL-I-UNMATEND error when it parses a declaration of an array of records. It does not associate the END with the record definition, and the resulting confusion in host variable scoping causes a fatal error.

To avoid the problem, declare the record as a type and then define your array of that type. For example:

```
    main.spa:

        program main (input,output);

        type
        exec sql include 'bad_def.pin';    !gives error
        exec sql include 'good_def.pin';   !ok
        var
            a : char;

        begin
        end.
----------------------------------------------------------------
    bad_def.pin

    x_record = record
    y  : char;
    variable_a:  array [1..50] of record
                a_fld1 : char;
                b_fld2  : record;
                            t : record
                                      v : integer;
                            end;
                end;
        end;
    end;
----------------------------------------------------------------

    good_def.pin

good_rec = record
        a_fld1 : char;
        b_fld2 : record
                t : record
                        v: integer;
                end;
        end;
end;

    x_record = record
        y  : char
        variable_a : array [1..50] of good_rec;
    end;
```

4.4.7 Supported PL/I Variable Declarations

The following list describes the variable declaration syntax that the SQL precompiler supports in PL/I:

- Declarations

  Declarations can include only the following PL/I data types:

  - CHARACTER

    CHARACTER can be abbreviated as CHAR.

  - CHARACTER VARYING

    CHARACTER VARYING can be abbreviated as CHAR VAR.

  - Date-time data types (Table 4–2 lists these data types.)
  - TINYINT

    TINYINT is FIXED BINARY(7).

  - FIXED BINARY, FIXED DECIMAL

    BINARY can be abbreviated as BIN, and DECIMAL can be abbreviated as DEC. Scale factors are not allowed on FIXED BINARY declarations.

  - FLOAT BINARY, FLOAT DECIMAL
  - SQL_DATE, SQL_DATE_ANSI, SQL_DATE_VMS
  - SQL_TIME, SQL_TIMESTAMP
  - SQL_INTERVAL (DAY TO SECOND)

    Use this data type for variables that represent the difference between two dates or times. (Table 4–2 lists all the supported INTERVAL data types.)

  - DECIMAL data type is converted to FIXED
  - NUMERIC data type is converted to PACKED

- Storage class attributes

  Any of the storage class attributes (BASED, AUTOMATIC, DEFINED, STATIC, variable, EXTERNAL, and INTERNAL) is allowed. The BASED attribute declarations must include a location reference.

- INITIAL attribute
- Structures

Structures are allowed without restriction.

- Arrays

  Arrays are permitted only for declarations of indicator arrays. Although you can use any data type for indicator array elements, Oracle Rdb recommends that you declare them as INTEGER variables.

  Multidimension array items are not supported. Arrays of structures are not supported. Arrays that are in a group that is itself an array are not supported. Dynamic-sized arrays are not supported.

The following example illustrates some PL/I declarations that SQL will and will not accept:

```
/* SQL will accept:
*/
DECLARE 1 E, (3 QE1, 3 QE2, 3 QE3) CHAR(10);

DCL P FIXED BIN(10), L FLOAT(53) BIN, K DECIMAL(10,2) FIXED;

DCL N VAR CHAR(10) INITIAL('XXXX');

DCL 1 B_P_REC   BASED(ADDR(S_P_REC)),
        2       PNUM    CHAR(6),
        2       PNAME   CHAR(20),
        2       WEIGHT  FIXED BIN(31),
        2       COLOR   CHAR(6),
        2       CITY    CHAR(10);

DCL D_IND_VEC (5) FIXED BIN(15) DEFINED(S_IND_VEC);

/* SQL will not accept:
*/
DCL A1 (1:10, 1:10) FIXED BIN(15);  /* multidimension table */

DCL B PICTURE '++++,+++,++9';       /* picture clauses */

DCL D1 BIT_FIELD(32);               /* bit fields */

DCL E1 FILE;                        /* file declarations */

DCL 1 F (10), 2 F1   FIXED BIN(15); /* arrays of structures */

DCL J POINTER;                      /* pointer declarations */

DCL K AREA(512);                    /* area declarations */

DCL L OFFSET(K);                    /* offset declarations */

DCL M FIXED BIN(31)                 /* external value declarations */
        EXTERNAL VALUE GLOBALREF;
```

Table 4–10 gives examples of PL/I variable declarations that SQL supports for each SQL data type.

Table 4–10 PL/I Declarations for SQL Data Types

SQL Example	PL/I Example
CHAR (10)	DCL STR1 CHAR(10);
VARCHAR (80)	DCL STR2 CHAR(80) VAR;
LONG VARCHAR	DCL STR3 CHAR(16383) VAR;
TINYINT	DCL NUM1 BIN FIXED(7);[1]
SMALLINT	DCL NUM1 BIN FIXED(15);[1]
INTEGER	DCL NUM2 BIN FIXED(31);[1]
BIGINT	DCL NUM3 BYTE_FIELD(8); or DCL NUM3 FIXED DEC(18);[2]
FLOAT (6) FLOAT (25)	DCL NUM4 BIN FLOAT(24); DCL NUM4 BIN FLOAT(53);
REAL	DCL NUM4 BIN FLOAT(24);
DOUBLE PRECISION	DCL NUM4 BIN FLOAT(53);
DATE	DCL P_DATE (SQL_DATE);
DATE ANSI	DCL P_DATE_A SQL_DATE_ANSI;
DATE VMS	DCL P_DATE_V SQL_DATE_VMS;
TIME	DCL P_TIME SQL_TIME(0);
TIMESTAMP	DCL P_TIMESTAMP SQL_TIMESTAMP(2);
INTERVAL DAY TO HOUR	DCL P_INTER1 SQL_INTERVAL (DAY TO HOUR);[3]
LIST OF BYTE VARYING	DCL STR4 CHAR(8);[4]

[1]PL/I does not support decimal scale factors on fixed binary data types; use the PL/I packed decimal data type to specify a scale factor.

[2]PL/I does not support BIGINTs. Use BYTE_FIELD(8) to pass BIGINTs to other languages; use FIXED DEC(18) (packed decimal) to work with BIGINTs in PL/I.

[3]Table 4–2 lists all the supported INTERVAL data types.

[4]PL/I does not support the LIST OF BYTE VARYING data type. This example shows how to retrieve the segmented string identifier, a pointer to the first element of the list, using an 8-byte character string. (You can use a BIGINT instead if you prefer.) To retrieve the values of individual elements of that list, use host language variables of data type CHAR or VARCHAR.

The online sample program sql_all_datatypes.spl provides examples of declaring variables and using them in SQL statements. The program also illustrates a variety of SQL data definition and data manipulation statements. After SQL is installed, you can print, type, or search the program to find sample code related to a variety of topics.

5
SQL Routines

This chapter describes routines used by SQL. All the routines described in this chapter can be called from any host language program that calls an SQL module or from any SQL precompiled program. These routines cannot be called from an SQL module.

Note

SQL defines all routines in uppercase on OpenVMS. Application programs must adhere to the rules about case-sensitivity of the language compiler to ensure that the programs call the routines correctly.

Table 5–1 describes the type of information that is presented in the following routine sections and the format used to present the information.

Table 5–1 Sections in the Routine Template

Section	Description
Routine Name	Appears at the top of the page
Overview	Appears below the routine name and explains, usually in one or two sentences, what the routine does
Format	Gives the routine entry point name and the routine argument list; also specifies whether arguments are required or optional
Returns	Gives the value returned from the routine
Arguments	Gives detailed information about each parameter

(continued on next page)

Table 5–1 (Cont.) Sections in the Routine Template

Section	Description
Description	Contains detailed information about specific actions taken by the routine, interaction between routine arguments, operation of the routine within the context of a specific operating system, and resources used by the routine
Usage Notes	Contains additional pieces of information related to application programming
Related Routines	Lists any related routines
Example	Shows an example using the routine

sql_close_cursors

Format

```
sql_close_cursors ()
```

Returns

No value returned.

Arguments

None.

Description

The sql_close_cursors routine closes all open cursors.

Usage Notes

- If you use the sql_close_cursors routine, you do not need to execute the CLOSE statement. This routine closes all open cursors.
- Use the sql_close_cursors routine to close cursors in any application that explicitly calls the DECdtm services. However, if you use default transaction support, you do not need to close any cursors because default transaction support closes all cursors for you.
- You can use the name sql$close_cursors to invoke this routine.

Related Routines

None.

Example

The following example shows an excerpt of an SQL precompiled program that uses the sql_close_cursors routine to close two cursors:

```
   .
   .
   .
/* Fetch records from two cursors.  The program has already declared them and
   opened them. */
EXEC SQL USING CONTEXT :CONTEXT_STRUC FETCH CURSOR_A;
EXEC SQL USING CONTEXT :CONTEXT_STRUC FETCH CURSOR_B;
   .
   .
   .
/* Close both cursors.*/
  sql_close_cursors();
   .
   .
   .
```

sql_deregister_error_handler

Deregisters an application's error handling routine

Format

```
sql_deregister_error_handler ()
```

Returns

No value returned.

Arguments

None.

Description

The sql_deregister_error_handler routine deregisters the application's currently registered error handling routine.

When you deregister a routine, SQL discontinues using the application's currently registered error handling routine. The standard error handling mechanisms are always in effect.

Usage Notes

- You do not have to use the sql_deregister_error_handler to deregister a routine before registering a new routine. The sql_register_error_handler routine deregisters the current routine and registers the new routine.

Related Routines

- sql_get_error_handler
- sql_register_error_handler

Example

See Example 5–1 for an example using the SQL error handling routines.

sql_get_error_handler

Gets the address of the application's currently registered error handling routine and the address of the user-specified data

Format

```
sql_get_error_handler (user-error-routine, user-data)
```

Returns

No value returned.

Arguments

user-error-routine
The address of an application's error handling routine

Value:	Address of an application's error handling routine
Data type:	Longword
Passing mechanism:	By reference

user-data
The address of the user-specified data

Value:	Address of the user-specified data
Data type:	Longword
Passing mechanism:	By reference

Description

The sql_get_error_handler routine gets the address of the application's currently registered error handling routine and the address of the user-specified data.

An application can use the sql_get_error_handler routine to get the address of the currently registered routine and user-specified data. The application can store the values in variables for use later in the program.

Related Routines

- sql_register_error_handler
- sql_deregister_error_handler

Example

See Example 5–1 for an example using the SQL error handling routines.

sql$get_error_text

Passes error text with formatted ASCII output to programs for processing

Format

```
sql$get_error_text (buf [,errmsglen])
```

Returns

The status code that results from the copy operation of the vector's text to the user's buffer.

Arguments

buf

The buffer declared to receive the text

Value:	Address of the buffer declared to receive the text
Data type:	Character string
Passing mechanism:	By descriptor

errmsglen

The number of characters allotted for the error messages to be returned. This parameter is optional.

Value:	Number of characters allotted for the error messages
Data type:	Word
Passing mechanism:	By reference

Description

Use the sql$get_error_text routine when you want to pass error text with formatted ASCII output (FAO) substitutions to your program for processing.

To use the sql$get_error_text routine, you must include a buffer (field) in your program declarations to receive the text SQL will pass to it. Declare this field as a text string with a length sufficient to accommodate the number of characters you expect for the message associated with the RDB$LU_STATUS value and for all follow-on messages. As an option, you can declare the buffer length as a separate field (defined as a signed word).

Usage Notes

- The status code returned by this routine is not the status code in the message vector.
- The following list shows the languages with which you can use the sql$get_error_text routine and how to call it from each language:
  - Ada

    ```
    procedure SQL_GET_ERROR_TEXT ( txt : out text-buffer-name;
                                   len : out short_integer );
    pragma INTERFACE (NONADA, SQL_GET_ERROR_TEXT):

    pragma IMPORT_PROCEDURE (INTERNAL => SQL_GET_ERROR_TEXT,
                             EXTERNAL => 'SQL$GET_ERROR_TEXT',
                             PARAMETER_TYPES => (text-buffer-name,
                                                 short_integer,)
                             MECHANISM =>(DESCRIPTOR, REFERENCE));
    ```

  - BASIC

    ```
    CALL SQL$GET_ERROR_TEXT(get_error_buffer)
    ```

  - C

    ```
    declaration of descriptor for text-buffer-name
    SQL$GET_ERROR_TEXT(&descriptor-name [, &text-buffer-length] )
    ```

  - COBOL

    ```
    CALL 'SQL$GET_ERROR_TEXT' USING BY DESCRIPTOR text-buffer-name
    [BY REFERENCE text-buffer-length]
    ```

  - FORTRAN

    ```
    CALL SQL$GET_ERROR_TEXT (%DESCR(text-buffer-name), [text-buffer-length])
    ```

  - Pascal

    ```
    type
        smallint = [word] -32768..32767;
    var
        buf : packed array [1..255] of char;
        len : smallint;

    procedure SQL$GET_ERROR_TEXT (
        var err_buff : [class_s] packed array [$L2..$U2:integer]
                       of char;
        var length : smallint); external;

    SQL$GET_ERROR_TEXT (buf, len);
    ```

- PL/I

```
DCL SQL$GET_ERROR_TEXT ENTRY (ANY, FIXED(15) BIN);
CALL SQL$GET_ERROR_TEXT (DESCRIPTOR(text-buffer-name)[,text-buffer-length]
```

- The sql$get_error_text routine returns a carriage return and line-feed character to separate follow-on messages from the primary message, and to separate follow-on messages from each other.

  The sql$get_error_text routine inserts the characters in the buffer declared to receive the text as delimiters between the messages. Typically, their presence eases display of the text to the terminal screen.

  However, if a program uses a forms product to display the message, the carriage return and line-feed characters are interpreted as unprintable characters.

  The following COBOL example shows one way to handle the presence of the carriage return and line-feed characters in the buffer:

```
CALL "SQL$GET_ERROR_TEXT" USING
        BY DESCRIPTOR BUFFER,
        BY REFERENCE LEN
     STRING CARRIAGE-RET, LINE-FEED DELIMITED BY SIZE INTO CRLF
     UNSTRING BUFFER DELIMITED BY CRLF INTO MSG-TXT_RDBFEL(1),
        MSG-TXT_RDBFEL(2), MSG-TXT_RDBFEL(3)
*
* CRLF is a PIC XX field that contains <cr><lf>.
* MSG-TXT-RDBFEL is an array of lines to be
* displayed for the error message.
*
```

Related Routines

- sql_get_error_text

Example

The following example shows the sql$get_error_text routine used in a C program:

```
/* This function uses the sql$get_error_text routine to display the
*  messages returned by various facilities for unexpected error conditions
*  that occur.  This program continues after these unexpected errors occur,
*  and allows the user to select the exit program option on the menu.

    void display_sqlget_message(void)

    {
```

```
char    get_error_buffer[301];
short    error_msg_len;

t_dsc.dsc$b_class = DSC$K_CLASS_S;
t_dsc.dsc$b_dtype = DSC$K_DTYPE_T;
t_dsc.dsc$w_length = 300;
t_dsc.dsc$a_pointer = (char *) (&get_error_buffer);
return_status = SQL$GET_ERROR_TEXT(&t_dsc,&error_msg_len);
get_error_buffer[error_msg_len] = '\0';
printf("\n\nThis condition was not expected.\n\n");
printf("%s",get_error_buffer);
release_screen = getchar();
printf("\n");

return;
}
```

sql_get_error_text

Passes error text with formatted ASCII output to programs for processing.

Format

```
sql_get_error_text (buf, len, errmsglen)
```

Returns

The status code that results from the copy operation of the vector's text to the user's buffer.

Arguments

buf
The buffer declared to receive the text.

Value:	Address of the buffer declared to receive the text
Data type:	Character string
Passing mechanism:	By reference

len
The length of the buffer declared to receive the text.

Value:	The length of the character string pointed to by the first parameter
Data type:	Longword
Passing mechanism:	By value

errmsglen
The number of characters allotted for the error messages to be returned.

Value:	Number of characters allotted for the error messages
Data type:	Longword
Passing mechanism:	By reference

Description

Use the sql_get_error_text routine when you want to pass error text with formatted ASCII output (FAO) substitutions to your program for processing.

To use the sql_get_error_text routine, you must include a buffer (field) in your program declarations to receive the text SQL will pass to it. Declare this field as a text string with a length sufficient to accommodate the number of characters you expect for the message associated with the RDB$LU_STATUS value and for all follow-on messages.

Usage Notes

- The status code returned by this routine is not the status code in the message vector.
- The following list shows the languages with which you can use the sql_get_error_text routine and how to call it from each language:
  - Ada

```
    procedure SQL_GET_ERROR_TEXT(ERROR_BUFFER     : out string;
                                 ERROR_BUFFER_LEN : in integer;
                                 ERROR_MSG_LEN    : out integer);
    pragma INTERFACE(SQL,SQL_GET_ERROR_TEXT);
    pragma IMPORT_PROCEDURE(internal => SQL_GET_ERROR_TEXT,
                            external => "SQL$GET_ERROR_TEXT",
                            parameter_types => (string,integer,integer),
                            mechanism => (reference,value,reference));

--  sql_get_error_text variables.
    ERROR_BUFFER     : string(1..256);
    ERROR_BUFFER_LEN : integer := 256;
    ERROR_MSG_LEN    : integer;

    sql_get_error_text(error_buffer,error_buffer_len,error_msg_len);
```

  - BASIC

```
external sub sql_get_error_text (string by ref, long by value, long by
ref)
MAP STRING error_buffer = 256
DECLARE LONG error_msg_len

CALL sql_get_error_text (error_buffer, LEN(error_buffer), error_msg_len)
```

– C

```
char    error_buffer[ n ];
int     error_msg_len;

sql_get_error_text(error_buffer, sizeof(error_buffer), &error_msg_len);
```

– COBOL

```
01 GETERRVARS.
   02  error-buffer-len         PIC S9(9) COMP VALUE 132.
   02  error-msg-len            PIC S9(9) COMP.
   02  error-buffer             PIC X(132).

CALL "sql_get_error_text" USING BY REFERENCE error-buffer,
                                BY VALUE error-buffer-len,
                                BY REFERENCE error-msg-len.
```

– FORTRAN

```
CHARACTER*256   error_buffer
INTEGER         error_msg_len

CALL sql_get_error_text (%REF(error_buffer),
1                        %VAL(LEN(error_buffer)),
2                        %REF(error_msg_len))
```

– Pascal

```
CONST
     error_buffer_len = 132;

VAR
     error_buffer  : packed array [1..error_buffer_len] of char;
     error_msg_len : integer;

PROCEDURE sql_get_error_text
        (%ref err_buffer: packed array [l1..u1:integer] of char;
         %immed err_buflen: integer;
         var err_length: integer); EXTERNAL;

sql_get_error_text (err_buffer := error_buffer,
                    err_buflen := LENGTH(error_buffer),
                    err_length := error_msg_len);
WRITELN (err_buffer: err_length);
```

– PL/I

```
DECLARE error_buffer           CHARACTER(256),
        error_buffer_len       FIXED BINARY(31) INITIAL (256),
        error_msg_len          FIXED BINARY(31);

CALL sql_get_error_text (error_buffer, error_buffer_len, error_msg_len)
```

- The sql_get_error_text routine returns a carriage return and line-feed character to separate follow-on messages from the primary message, and to separate follow-on messages from each other.

  The sql_get_error_text routine inserts the characters in the buffer declared to receive the text as delimiters between the messages. Typically, their presence eases display of the text to the terminal screen.

  However, if a program uses a forms product to display the message, the carriage return and line-feed characters are interpreted as unprintable characters.

  The following COBOL example shows one way to handle the presence of the carriage return and line-feed characters in the buffer:

```
*
* CRLF is a PIC XX field that contains <cr><lf>.
* MSG-TXT-RDBFEL is an array of lines to be displayed for the error message.
*
    STRING CARRIAGE-RET, LINE-FEED DELIMITED BY SIZE INTO CRLF.
    CALL "sql_get_error_text" USING BY REFERENCE BUFFER,
                                    BY VALUE BUF_LEN
                                    BY REFERENCE MSG_LEN.
    UNSTRING BUFFER DELIMITED BY CRLF INTO MSG-TXT_RDBFEL(1),
        MSG-TXT_RDBFEL(2), MSG-TXT_RDBFEL(3).
```

Related Routines

- sql$get_error_text

Example

The following example shows the sql_get_error_text routine used in a C program:

sql_get_error_text

```
/*
*  This function uses the sql_get_error_text routine to display the
*  messages returned by various facilities for unexpected error conditions
*  that occur.  This program continues after these unexpected errors.
*/
void display_sqlget_message(void)
   {
    char  err_buf[ 1024 ];
    int   err_msg_len;

    sql_get_error_text(&err_buf, sizeof(err_buf), &err_msg_len);
    err_buf[err_msg_len] = 0;
    printf("%s\n",err_buf);
    return;
   }
```

sql_get_message_vector

Retrieves information from the message vector about the status of the last SQL statement.

Format

```
sql_get_message_vector (addr, index)
```

Returns

No value returned.

Arguments

addr
Address of a variable into which the requested vector element will be written.

Value:	Address of the variable declared to receive the vector element
Data type: longword (32 bit)	
Passing mechanism:	By reference

index
The index value of the vector element to return.

Value:	A value greater than or equal to 1 and less than or equal to 20
Data type:	Unsigned longword
Passing mechanism:	By value

The following table shows the index values and how they map to vector elements and the information contained in each vector element.

Index Value	Information Returned
1	Number of arguments in the vector
2	Primary status code of the last SQL statement
3	Number of FAO arguments to primary message
4–20	Return status for follow-on messages, if any

sql_get_message_vector

Description

Use the sql_get_message_vector routine to retrieve information from the RDB$MESSAGE_VECTOR array. The array provides information about the execution status of SQL statements. Index 2 of the sql_get_message_vector routine returns the primary status code of the last SQL statement. This index is comparable to the SQLCODE status parameter.

The ANSI/ISO SQL standard does not include the sql_get_message_vector routine or the RDB$MESSAGE_VECTOR array. For application programs that comply with the standard, use the SQLCODE or SQLSTATE status parameters. Furthermore, the status values returned for a particular condition may change in future versions of Oracle Rdb.

The status values that are stored in the message vector are intended to be supplementary information to the status parameters SQLCODE and SQLSTATE. Use the sql_get_message_vector routine if the information provided by SQLCODE or SQLSTATE is ambiguous and your application needs a more specific error code to handle the error condition.

Table 5–2 shows the relationship between the indexes returned by sql_ message_vector and the fields in RDB$MESSAGE_VECTOR.

Table 5–2 Relationship Between sql_message_vector and RDB$MESSAGE_ VECTOR

sql_message_vector Indexes	RDB$MESSAGE_VECTOR Fields
Index 1	RDB$LU_NUM_ARGUMENTS
Index 2	RDB$LU_STATUS
Index 3	RDB$LU_ARGUMENTS
Index 4–20	Return status for follow-on messages, if any

Usage Notes

- The following list shows the languages with which you can use the routine sql_get_message_vector and how to call it from each language:
  - Ada

```
procedure sql_get_message_vector (buffer_name : out ;
                                  index: in );
pragma INTERFACE (NONADA, sql_get_message_vector):

pragma IMPORT_PROCEDURE (INTERNAL => sql_get_message_vector,
                         EXTERNAL => "sql_get_message_vector",
                         PARAMETER_TYPES => (integer,integer)
                         MECHANISM =>(REFERENCE, VALUE));
```

– BASIC

```
external sub sql_get_message_vector(long by ref, long by value)
call sql_get_message_vector(buffer_name, index)
```

– C

```
int buffer_name;
int index;

sql_get_message_vector(&buffer_name, index);
```

Declaring the arguments as the int data type ensures that the correct data type is used for all platforms.

– COBOL

```
CALL 'sql_get_message_vector' USING BY REFERENCE buffer-name
                              BY VALUE index
```

– FORTRAN

```
CALL sql_get_message_vector (buffer-name, index)
```

– Pascal

```
var
    buffer, index : integer;

procedure sql_get_message_vector (
    var buffer: integer;
    index : [immediate,readonly] integer ); external;

sql_get_message_vector (buffer, index);
```

– PL/I

```
DCL sql_get_message_vector ENTRY (ANY, FIXED(31) BIN);
CALL sql_get_message_vector (REFERENCE(buffer-name),index)
```

Related Routines

None.

Example

The following example shows an excerpt of a C program that calls an SQL module and uses the sql_get_message_vector to return the status of the SQL statement:

```
   .
   .
   .
/* Error handler, using sql_get_message_vector. */

get_msgvec( )

{
int index;
int status_code;
int arg_cnt;

/* Declare the literal for contraint violation status. */
int RDB$_INTEG_FAIL;

/* Get the message vector argument count. */

index = 1;
sql_get_message_vector(&arg_cnt, index);

/* Get the status code. */

index = 2;
sql_get_message_vector(&status_code, index);

  if (status_code == RDB$_INTEG_FAIL)
     printf("Constraint violation. ");
     printf("You are trying to insert a department code\n");
     printf("which already exists in the table.");
     exit(1);

/* You can also check for the follow-on arguments, if the arg_cnt is greater
*  than 1.
*/
}

main( )
{
   .
   .
   .
```

```
  insert_data (&SQLCODE, department_code, department_name, manager_id);
  if (SQLCODE != 0)
   get_msgvec();
}
```

sql_register_error_handler

Registers an application's error handling routines with the SQL precompiler.

Format

```
sql_register_error_handler (user-error-routine, user-data)
```

Returns

No value returned.

Arguments

user-error-routine

The address of an application's error handling routine

Value:	Address of an application's error handling routine
Data type:	Address
Passing mechanism:	By value

user-data

The address of the user-specified data

Value:	Address of the user-specified data
Data type:	Address
Passing mechanism:	By value

Description

The sql_register_error_handler routine registers the application's error handling routine with SQL. When SQL determines that it will return a negative value for SQLCODE, SQL calls the error handling routine that is currently registered. The standard error handling routines are always in effect, whether special error handling routines have been registered or not. After the error handling routine executes, control returns to SQL.

An application can contain and call more than one error handling routine. However, only one routine can be active at a time.

An application can use the sql_get_error_handler routine to store the address of a registered routine for use later in the program.

To deregister a routine, use the sql_deregister_error_handler routine.

For information about declaring the error handling routines in the supported programming languages, see the *Oracle Rdb Guide to SQL Programming*.

Usage Notes

- The application's error handling routine must accept four parameters. The following three parameters are passed by reference: RDB$MESSAGE_VECTOR, SQLCODE, SQLSTATE. The fourth parameter is the address of the user-specified data and is passed by reference.
- If you call more than one error handling routine, SQL uses the most recently registered routine.

Related Routines

- sql_deregister_error_handler
- sql_get_error_handler

Example

Example 5–1 shows how to use the SQL error handling routines in a precompiled C program.

Example 5–1 Using SQL Error Handling Routines

```
/* This program demonstrates the use of the SQL error handling routines,
*  sql_register_error_handler, sql_deregister_error_handler, and
*  sql_get_error_handler.   Although the use of the sql_get_error_handler
*  routine is not necessary in this simple program, it is included here
*  to demonstrate how to use the routine to store the address of the
*  currently registered routine and the address of user data in variables.
*/

#include <sql_literals.h>

/* Definition of rdb$message_vector. */
typedef struct {
   long RDB$LU_NUM_ARGUMENTS;
   long RDB$LU_STATUS;
   long  RDB$LU_ARGUMENTS[18];
   } RDB$MESSAGE_VECTOR;

/* Definition of structure to hold user data. */
```

(continued on next page)

Example 5–1 (Cont.) Using SQL Error Handling Routines

```
typedef struct {
   char sql_proc_name[31];
   char sql_col_value[31];
          } err_struct;

/* Error handling routine for constraint violations.  This routine traps
*  constraint violations and prints out an error message.
*/

static
  void dupl_error_handler(
       RDB$MESSAGE_VECTOR *msgvec,
       int                *sqlcode,
       char               *sqlstate,
       void               *user_info)
/* The preceding declaration for sqlcode refers to the internal sqlcode value,
* a 32-bit quantity.
*/
    {

     err_struct  *my_info;
     my_info = (err_struct *)user_info;

     if ((*sqlcode == SQLCODE_INTEG_FAIL) &&
        ((strcmp(my_info->sql_proc_name, "INSERT_JOBS")) == 0))
      {
      printf(" The Job Code %s is already in use.\n", my_info->sql_col_value);
      }

/* You can add more conditional statements to this error procedure to handle
*  errors from several SQL statements.
*/

    }

/* Error handling routine for errors that occur when you start a transaction.
*  This routine prints out an error message.
*/

static
  void txn_error_handler(
       RDB$MESSAGE_VECTOR *msgvec,
       int                *sqlcode,
       char               *sqlstate,
       void               *user_info1)
    {

     if ((*sqlcode == SQLCODE_DEADLOCK) || (*sqlcode == SQLCODE_BAD_TXN_STATE)
           || (*sqlcode == SQLCODE_LOCK_CONFLICT))
```

(continued on next page)

Example 5–1 (Cont.) Using SQL Error Handling Routines

```
       printf("Unable to start a transaction. \n");
    }

main( )
{
/* Variables used by the main program. */
    void    (*rtn_ptr)();
    err_struct  *err_struct_ptr = NULL;

    char    j_code[5];
    char    w_class[2];
    char    j_title[21];
    char    release_screen;

/* Define the SQLCA. */
    EXEC SQL INCLUDE SQLCA;

/* Initialize user-defined information. */
    err_struct err_s = {" ", " "};

/* Declare the database. */
    EXEC SQL DECLARE ALIAS FILENAME 'personnel';

/* Register the first error handling routine. */
    sql_register_error_handler(txn_error_handler,0);

/* Store the address of the currently registered pointer in a variable. */
    sql_get_error_handler(&rtn_ptr, &err_struct_ptr);

    printf("Please enter the Job Code (or EXIT):\n");
    scanf(" %s", j_code);
    release_screen = getchar();

    while (((strcmp(j_code,"exit")) != 0) &&
          ((strcmp(j_code,"EXIT")) != 0))
     {
```

(continued on next page)

Example 5–1 (Cont.) Using SQL Error Handling Routines

```
      printf("Enter the Wage Class:  ", w_class);
      scanf(" %s", w_class);
      release_screen = getchar();
        while  (((strcmp(w_class,"1")) != 0) &&
               ((strcmp(w_class,"2"))  !=0) &&
               ((strcmp(w_class,"3"))  !=0) &&
               ((strcmp(w_class,"4"))  !=0))
         {
          printf("Please enter one of the following values for Wage Class:\n");
          printf(" 1  2  3  4\n");
          scanf(" %s", w_class);
          release_screen = getchar();
         }

      printf("Please enter the Job Title: \n");
      scanf(" %s", j_title);
      release_screen = getchar();

/* Start a transaction. */

      EXEC SQL SET TRANSACTION READ WRITE NOWAIT
                   RESERVING JOBS FOR EXCLUSIVE WRITE;


/* Register the second error handling routine. */
      sql_register_error_handler(dupl_error_handler, &err_s);

/* Store information in a structure for use by the error handling routine. */
      strcpy(err_s.sql_proc_name, "INSERT_JOBS");
      strcpy(err_s.sql_col_value, j_code);

      EXEC SQL INSERT INTO JOBS
      (JOB_CODE, WAGE_CLASS, JOB_TITLE)
      VALUES
      (:j_code, :w_class, :j_title );

      if (SQLCA.SQLCODE == SQLCODE_SUCCESS)
        EXEC SQL COMMIT;
      else
        EXEC SQL ROLLBACK;


/* Deregister the error handling routine. */
      sql_deregister_error_handler();

      printf("Please enter the Job Code (or EXIT):\n");
      scanf(" %s", j_code);
      release_screen = getchar();
```

(continued on next page)

Example 5–1 (Cont.) Using SQL Error Handling Routines

```
/* Register the txn_error_handler routine again.  Use the address stored in
*   rtn_ptr.
*/
      sql_register_error_handler(rtn_ptr, 0);

     }

    return;

}
```

sql_signal

Format

```
sql_signal ()
```

Returns

No value returned.

Arguments

None.

Description

The sql_signal routine signals to your program condition handler an error that occurs on the execution of an SQL statement. If your program does not contain a condition handler, the sql_routine prints an error message and exits the program when an error occurs.

If the host language compiler does not provide a signaling mechanism, you must provide your own condition handler.

Usage Notes

- You can use the name sql$signal to invoke this routine as well as sql_signal.
- The following list shows the languages with which you can use sql_signal routine and how to call it from each language:
  - Ada

    ```
    procedure SQL_SIGNAL
    pragma INTERFACE (NONADA,SQL_SIGNAL)
    pragma IMPORT_PROCEDURE (SQL_SIGNAL,"sql_signal")
    ```

    ```
    sql_signal;
    ```

    See your Ada documentation for information about using calls that signal errors.

– BASIC

```
CALL sql_signal();
```

See your BASIC documentation for further discussion of creating a condition handler.

– C

```
sql_signal();
```

C for OpenVMS provides a run-time library routine, VAXC$ESTABLISH, that you use to create a program condition handler. See your C documentation for further discussion of creating a condition handler.

– COBOL

```
CALL "sql_signal"
```

COBOL for OpenVMS is a language that automatically establishes a condition handler for you. Therefore, unless your program has called routines to establish another condition handler, calling sql_signal causes your COBOL program to display messages and then continue program execution under control of the COBOL condition handler. See your COBOL documentation for a discussion of the COBOL condition handler.

– FORTRAN

```
CALL sql_signal
```

FORTRAN for OpenVMS is a language for which you either call LIB$ESTABLISH to create a program condition handler or you rely on the OpenVMS condition handler. See your FORTRAN documentation for further discussion of creating a program condition handler.

– Pascal

In Pascal programs, you must declare sql_signal as an external procedure before calling the routine.

```
procedure sql_signal; external;
sql_signal;
```

Pascal provides a routine you can use to establish an OpenVMS condition handler. See your Pascal documentation for further discussion of creating a program condition handler.

– PL/I

In PL/I programs, you must declare sql_signal as an external entry before calling the routine.

```
DCL sql_signal EXTERNAL ENTRY;
CALL sql_signal;
```

PL/I is a language that automatically establishes a condition handler in programs. Therefore, unless your program has called routines to establish another condition handler, calling sql_signal will cause your PL/I program to display messages and then continue program execution under control of the PL/I condition handler. See your PL/I documentation for a discussion of the PL/I condition handler.

Related Routines

None.

Example

The following excerpt from the SQL$REPORT.SC sample program shows how to call the sql_signal routine:

```
/* Main loop */
do
    {
    /* FETCH by SQL to get a database record */
    EXEC SQL FETCH REPORT_CURSOR INTO
                :employee_id, :last_name, :first_name,
                :job_code, :department_code, :salary_amount;

    /* Check return status and take appropriate action */
    switch (SQLCA.SQLCODE)
        {
        /* If a record was returned, print a detail line */
        case SQL_SUCCESS :
            detail_line();
            break;
        /* If end of stream is encountered, print the final totals */
        case STREAM_EOF :
            job_code_foot();
            dept_code_foot();
            final_foot();
            break;
        /* Any other status is an error condition and will be trapped by the
           SQL error handler */
        default :
            break;
        }
    }
while (SQLCA.SQLCODE == SQL_SUCCESS);
```

```
/* Close the report file */
fclose(report_file);

/* Close the cursor */
EXEC SQL CLOSE REPORT_CURSOR;

/* Rollback the transaction */
EXEC SQL ROLLBACK;

exit(1);

ERROR_HANDLER:
    printf("\nAn unexpected error was encountered %d", SQLCA.SQLCODE);
    sql_signal();
}
```

Index

B

C

D

E

J

K

L

M

N

Q

R

S

T

U

V

W

Y