# Personal Oracle Lite™ Programmer's Guide

Release 2.4

Part No. A52614-03

ORACLE®

Enabling the Information Age

Personal Oracle Lite Programmer's Guide, Release 2.4 Beta

Part No. A52614-03



Contributing Authors: Workgroup Products Documentation

# Preface

This preface introduces the *Personal Oracle Lite Programmer's Guide*. It describes:

- what you need to know to use this manual
- the manual's chapters and appendices
- typographical conventions used in this manual
- where to find more information
- where to send your comments

## Audience

The *Personal Oracle Lite Programmer's Guide* is written for programmers who want to create powerful client/server applications that:

- are optimized for small-footprint devices
- can access and manipulate SQL data from within an object-oriented GUI
- can query instances of a C++ class as if they were rows in a SQL table
- run under Windows 3.1, NT 3.51 or Windows 95

This manual describes Personal Oracle Lite's relational database capabilities.

## How This Manual Is Organized

We divide this manual into the following chapters:

| | |
|---|---|
| **Chapter 1: Introduction** | Introduces Personal Oracle Lite. |
| **Chapter 2: SQL Overview** | Provides an overview of SQL and its functions. |
| **Chapter 3: Data Types and Literals** | Describes the supported Personal Oracle Lite data types and literals. |
| **Chapter 4: SQL Functions, Predicates and Expressions** | Describes SQL functions, expressions and predicates in detail. |
| **Chapter 5: Embedded SQL C Binding** | Describes the procedure for embedding SQL in a C program. |
| **Chapter 6: ODBC Binding** | Describes ODBC and its functions. |
| **Chapter 7: ODBC Information Types** | Lists the ODBC information types. |
| **Chapter 8: System Catalog** | Lists the object types in the system catalog |
| **Chapter 9: Replication** | Describes how to create replication applications |
| **Chapter 10: Database Utilities** | Describes how to use the database utilities |

## Conventions Used in This Manual

Note these typographical conventions when reading this manual:

| | |
|---|---|
| `Monospace text` | Type the text exactly as shown. Monospace text represents commands or SQL statements. |
| `lowercase mono` | Lowercase characters within command lines represent variables; substitute an appropriate value for the variable. In examples, lowercase characters represent sample values for the variables. |
| *lowercase italics* | Lowercase italics in the text represent variables. Substitute an appropriate value for the variable. |
| [ ] | Brackets enclose optional items or indicate a function key. Do not enter the brackets. |
| \| | A vertical bar represents an “or” option among several options. You enter only one of the options. Do not enter the vertical bar. |
| { } | Braces represent a set of choices where you must choose at least one of the options allowed. |
| Punctuation | Punctuation other than brackets and vertical bars must be entered in commands exactly as shown. |
| UPPERCASE | Uppercase characters within the text represent command names, SQL reserved words and keywords, and file names. |
| \DIRECTORY | A backslash before a directory name indicates that it is a subdirectory. |
| C:\> | C:\> represents the operating system command prompt on the current network or hard disk drive. If your drive letter designation is different, your prompt will reflect that letter. |

## Your Comments Are Welcome

We value your comments as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of this manual is a Reader's Comment Form. We encourage you to use it to tell us what you like and dislike about this (or other) Oracle manuals.

If the form is missing, or if you would like to contact us, write us at:

**Workgroup Products Documentation Manager**
**Oracle Corporation**
**500 Oracle Parkway**
**Box 659107**
**Redwood Shores, CA 94065**

# Contents

CHAPTER

1

# Using Personal Oracle Lite

This chapter introduces Personal Oracle Lite and provides an overview of the tools and interfaces that make up Personal Oracle Lite, including:

- introduction to Personal Oracle Lite
- description of the Personal Oracle Lite product package
- how to get started with Personal Oracle Lite
- where to get more information about Personal Oracle Lite

For information on installing and configuring Personal Oracle Lite, see the *Personal Oracle Lite Installation Guide* CD-ROM insert.

## What is Personal Oracle Lite?

Personal Oracle Lite is a lightweight, object/relational database that is the ideal solution for mobile application deployment. Designed for mobile and other small-footprint devices, Personal Oracle Lite runs in under 1 megabyte of memory, and can be installed in as little as 3megabytes of hard disk space.

The Personal Oracle Lite development environment supports seamless interoperability on objects across both SQL and object databases. Use Personal Oracle Lite to create compact mobile applications that can access and manipulate SQL data from within an object-oriented GUI, or query instances of a C++ class as if they were rows in a SQL table. Personal Oracle Lite C++ applications can be designed to run under either:

- Windows 3.1x
- Windows NT 3.51 and higher
- Windows 95

Personal Oracle Lite provides the following four development interfaces:

- For relational database development:
  - ODBC
  - Embedded SQL
- For object database development:
  - ODMG C++ binding
  - Object kernel API (OKAPI)

These interfaces can be used either independently or in combination and are described below.

### Relational Database Interfaces

Personal Oracle Lite supports traditional relational database development through two SQL options, both of which are described in this Guide.

- **Embedded SQL --** Personal Oracle Lite comes with a SQL-92 compliant SQL interface, enabling you to embed SQL statements within a C or C++ program.
- **ODBC --** Microsoft's Open Database Connectivity Interface (ODBC) specifies a set of functions that allow connections to databases, preparation and execution of SQL statements at runtime, and application retrieval of query results. ODBC is widely supported by both database and (front-end tool) vendors. This

provides programmers with a wide range of options for creating Personal Oracle Lite applications.

## Object Database Development Interfaces

Personal Oracle Lite supports the following object-oriented interfaces:

- **ODMG C++ binding --** ODMG is an industry standard for object databases defined by the Object Database Management Group. ODMG C++ binding enables programmers to store and retrieve disk-resident objects through a library of persistent C++ collection classes. ODMG C++ binding also supports some facilities that make it suitable for use with SQL, such as:
  - the ability to query a collection of objects
  - the ability to query a direct SQL interface
  - transaction management capability
- **Object Kernel API (OKAPI) --** The application programming interface (API) to Personal Oracle Lite's object kernel includes support for the following important database features:
  - runtime class creation and access to class information
  - direct object access based on object identity and navigation
  - object clustering and grouping
  - queries on classes and their subclasses
  - object naming and inter-object relationships
  - binary large object (BLOB) data
  - transaction and crash recovery

## Personal Oracle Lite Product Package

Personal Oracle Lite comes on a single CD that supports both 16-bit (Windows 3.1x) and 32-bit (Windows NT 3.51 or higher and Windows 95) development. The product package consists of the following components:

| Component | Description |
|---|---|
| **Personal Oracle Lite Database** | Lightweight object/relational database with replication capability. Personal Oracle Lite can participate in the symmetric replication environment of an Oracle 7 server, enabling remote bi-directional replication. |
| **Navigator** | Graphical user interface that enables you to manage databases, database objects, and database connections.<br>*The Navigator is available for Personal Oracle Lite on Windows 95 or NT, only.* |
| **Data Manager** | Utility that enables you to move data and perform other database management tasks.<br>*The Data Manager is available for Personal Oracle Lite on Windows 3.1x, only.* |
| **SQL*Net** | Networking software that provides the connectivity to remote Oracle 7 servers. Through SQL*Net Version 2, this release of Personal Oracle Lite includes support for the TCP/IP, SPX, and Named Pipes network protocols. |
| **SQL*Plus** | An ad hoc query and reporting tool. |
| **Documentation** | The following documentation is provided:<br>• context-sensitive online Help for the Navigator, available directly from the Navigator console<br>• online manuals (this manual and the SQL*Net and SQL*Plus manuals, are all viewable online with Adobe Acrobat), located in the Personal Oracle Lite Program Group.<br>• online Release Notes, also located in the Personal Oracle Lite Program Group.<br>Note: All online documentation can be printed by the page, chapter, or document. |

## Getting Started With Personal Oracle Lite

When you install Personal Oracle Lite, an ODBC data source name, POLITE, is created, and a starter database POLITE.ODB is dedicated. The location of new databases for the data source name POLITE is set to *Oracle_Home*\POLDB, where *Oracle_Home* is:

- ORAWIN95 for Windows 95
- ORANT for Windows NT
- ORAWIN for Windows 3.1x and Windows for Workgroups

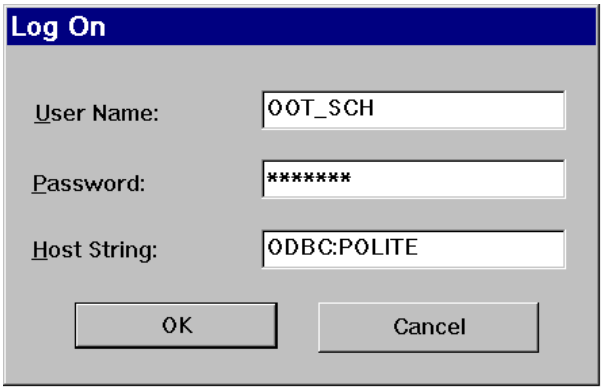
A default user, OOT_SCH, is also set up for you during installation. In Personal Oracle Lite, users are the same as schemas. You can use this default user name until you establish user names of your own.

### Connecting to the Starter Database

To access the starter database, POLITE.ODB, you must first connect to it.

#### Connecting from an Oracle Application

To connect to the starter database using an Oracle application such as SQL*Plus or Oracle Forms, enter OOT_SCH as the user name and `ODBC:POLITE` as the host string. The password is required by SQL*Plus but can be any string. It is not checked or saved because the database is not password protected.




#### Connecting from an ODBC Application

Use the default ODBC data source name, POLITE, as the data source name (DSN) when connecting to the starter database from an ODBC application.

## Creating a New Database

When you create a new database using the POLITE data source name, the new database file is located in the *Oracle_Home*\POLDB directory. For ease of maintenance, it is recommended that you use one database directory for all of your databases. You can create a new database using the Navigator or a command line utility.

### Using the Navigator

To create a new database using the Navigator:

1. Start the Navigator if it is not already running.
2. Right-click the Oracle Lite Databases folder.
3. Click New. A new database property sheet appears.
4. Type the name of the new database. Personal Oracle Lite creates the database and automatically connects to it.

All database files created using the Navigator are created in the *Oracle_Home*\POLDB directory and are associated with the POLITE data source name.

### Using a Command Line Utility

To create a new database from the command line, use the CREATEDB utility: For example:

```
CREATEDB polite dbname
```

where *dbname* is the name you want to give to the new database. For more information on the CREATEDB utility, see Chapter 9.

**Note:** For simple installations, Oracle recommends that you use the Navigator to create all database files under the POLITE data source name. The Navigator records in the Windows registry all files it creates, but does not register files created from the command line or under other ODBC data source names.

## Creating a New ODBC Data Source Name

Through the ODBC Administrator utility, you can create additional ODBC data source names and assign to them Personal Oracle Lite database files you create. For instruction on how to use the ODBC Administrator utility, see Chapter 9.

When you create a new ODBC data source name, the ODBC Administrator utility creates an entry in your ODBC.INI file (and the registry on Windows 95 or NT) which specifies the following:

- the data source name

- a description of the database contents
- the data directory in which all files associated with the data source name will be created
- the database file name (without the extension .ODB)

For example, the DSN entry for POLITE in the ODBC.INI file might contain the following:

```
[POLITE]
Description=Oracle Lite Data Source
DataDirectory=C:\ORAWIN\POLDB
Database=POLITE
```

## Connecting to a New Database

You can connect to a new database either through Oracle applications or through ODBC-based applications.

### From an Oracle Application

To connect to a new database using an Oracle application such as SQL*Plus, use the following connect string:

```
ODBC:POLITE:dbname
```

where *dbname* is the name of the new database.

You may also replace `POLITE` with any other previously defined ODBC data source name. If there is more than one database associated with a data source name, use the format:

```
ODBC:dsn:dbname
```

where *dsn* is the name of the data source and *dbname* is the name of the database.

### From an ODBC-based Application

Once you have assigned a data source name, you can connect to the new database using either of these methods:

1. To connect to a database with a data source other than POLITE, specify the data source name. For example, if you create data source named `MYDSN`, use `MYDSN` as the data source name in the ODBC application, and you will automatically connect to the default database for that data source name.
2. As with Oracle applications such as SQL*Plus, you can use the default data source name and still connect to a new database. For example, connect to the

POLITE data source name, then issue the following command to switch to the MYDB database:

```
SET CATALOG MYDB
```

To verify which database you are connected to, use the ODBC function for getting the database name. For example:

```
SELECT CAST({FN DATABASE()} AS CHAR(16)) FROM DUAL;
```

**Note:** The ODBC functions SQLConnect() and SQLDriverConnect() require a user name, but the password can be NULL or an empty string.

## Creating New Users

Since Personal Oracle Lite is a single-user database, it is assumed that if you have access to the operating system on which it is running, then you have complete access to the database. Thus, certain concepts used in a multiple-user database, such as roles and passwords, are not applicable.

In Personal Oracle Lite, users are the equivalent of schemas. In other words, a user is a named group of database objects. Although designed as a single-user environment, Personal Oracle Lite enables you to create multiple users so that you can partition database objects within an application. Unlike multiple-user databases, a user is not a mechanism for security. Any object created in the database is accessible to all users in the database.

When you first install Personal Oracle Lite, a default user, OOT_SCH is created for you. You can use this default user until you establish user names of your own, if needed. Once you have connected to Personal Oracle Lite as the default user name OOT_SCH, you can create new users.

You can create new users through the Navigator by right-clicking on the User folder and selecting New. Alternatively, you can use SQL*Plus to connect and then create a new user with the command:

```
CREATE SCHEMA schema_name
```

where *schema_name* is the name of a new user.

Then, you must issue a COMMIT command before you can connect with the new user name.

You may change the current default user by issuing the following SQL command from an ODBC-based application:

```
SET SCHEMA schema_name
```

From SQL*Plus, you can issue the CONNECT command to reconnect to the database using the new user name. For example:

```
CONNECT SCOTT/TIGER@ODBC:POLITE:DB1
```

will connect to the `DB1.OBD` database as user `SCOTT`.

Since each Personal Oracle Lite database is self-contained, a user created in one database is not available in any other database until it is created there. When adding database files to a data source name, such as POLITE, you may want to create the new user in both the default database file (the one specified in the ODBC.INI "Database" parameter) as well as the new database file.

## Copying Tables

You can copy tables by dragging and dropping them using the Navigator (on Windows 95 or Windows NT) or Data Manager (on Windows 3.1x). For instructions, see the online help for these tools.

## Backing Up a Database

Since the Personal Oracle Lite database occupies only one file, you can back up a database file by copying it to a new location. You can back up a database using the Navigator (on Windows 95 or Windows NT) or Data Manager (on Windows 3.1x). For instructions, see the online help for these tools. To back up a database from your operating system, follow the instructions, below:

On Windows 95 or Windows NT 4.0:

1. From the Explorer, go to either the `orawin95\poldb` directory or the `orant\poldb` directory and highlight the database you wish to back up (for example, `MYDB.ODB`).
2. Choose Copy from the Edit menu.
3. In the directory where you want to keep the backed-up copy, choose Paste from the Edit menu.

On Windows 3.1x or Windows NT 3.51:

1. From the File Manager, go to either the `orawin\poldb` directory or the `orant\poldb` directory and highlight the database you wish to back up (for example, `mydb.odb`).
2. Choose Copy from the File menu. Enter a new path and filename, and click OK.

## Using Sample Applications

If you choose the Complete installation or select the Software Developer Kit from the Custom installation menu, sample applications demonstrating how to use Personal Oracle Lite will be installed in your *Oracle_Home*\Lite\Examples subdirectory for the following tools:

| Tool | Location of Sample Applications |
|---|---|
| Power Objects | *Oracle_Home*\Lite\Examples\OPO |
| Visual Basic | *Oracle_Home*\Lite\Examples\VB |
| Access | *Oracle_Home*\Lite\Examples\Access |
| Delphi | *Oracle_Home*\Lite\Examples\Delphi\Delphi1<br>*Oracle_Home*\Lite\Examples\Delphi\Delphi2 |
| Power Builder | *Oracle_Home*\Lite\Examples\PB |
| Embedded SQL | *Oracle_Home*\Lite\Examples\ESQL |

### Working with Oracle Tools

The Open Client Adapter (OCA) is supplied with Personal Oracle Lite as UB73Wxx.DLL so that Oracle tools such as SQL*Plus may run with our ODBC driver. Other tools, such as Forms and Reports also use the OCA. For the latest information on using Personal Oracle Lite with Developer/2000, please refer to the documentation on the Open Client Adapter (OCA) supplied with Developer/2000 version 1.3.2 as POLINFO.PDF. The Open Client Adapter must be installed (or reinstalled) after you install Personal Oracle Lite.

**Note:** The error "UB-30017: ODBC Data Source not available" when attempting to connect using an Oracle tool like SQL*Plus is commonly caused by attempting to connect as a user who does not exist.

### Working with Oracle Power Objects

Oracle recommends that you upgrade to Oracle Power Objects version 2.1, which includes Personal Oracle Lite release 2.3.0.0.26. If you are using Oracle Power Objects, please contact Oracle WorldWide Customer Support about upgrading to version 2.1.

If you have installed Oracle Power Objects version 2.0, it will already have installed Personal Oracle Lite version 2.3.0.0.13 in a directory called \DBS. To use this version of Personal Oracle Lite in conjunction with Power Objects 2.0:

1. Install Personal Oracle Lite normally under the *Oracle_Home* directory
2. Make sure that *Oracle_Home*\BIN appears in the PATH.

   Users of Windows 3.1x should make sure *Oracle_Home*\BIN appears before the OPO20\DBS directory, and should use the CREATEDB utility to create files.

Be aware that the ODBC Driver Manager will connect using the latest OOTODxx.DLL found in the \Windows \Win95 or \Winnt system directory, and that it must find the other DLLs for version 2.4 using the search path before it sees the old 2.3.0.0.13 version in the \DBS directory.

**Note:** Power Objects 2.0 users may find that they can't drag tables between a 2.3.0.0.13 database and a 2.4 database. The ODBC driver manager may report the error "Data source rejects establishment of connection." This problem may be solved by changing the driver for existing DSNs from "OPO Local Database Driver" to the new "Oracle Lite ODBC Driver" using the ODBC Administrator.

## Working with Visual Basic and Other Tools

Microsoft's Visual Basic and most other ODBC-based tools require a primary key or unique index in order to update a table through a data control. This will prevent most tools from updating a view, since the underlying database may not report an index on the view's base table. In Visual Basic 4.0 this shows up as a failure on macros such as NEW, EDIT, DELETE, etc., and bound data controls. A workaround is to code SQL directly in the Visual Basic procedures. For an example, see the Visual Basic sample application provided with Personal Oracle Lite.

## Working with PowerBuilder

When PowerBuilder connects to a Personal Oracle Lite database a message may appear stating that "Catalog tables could not be created and are not available for use." This message can be ignored, as the tables are created the first time PowerBuilder connects to a Personal Oracle Lite database. However, the process may take several seconds.

When creating the Database Profile, the DBPARM connect option in the Database Profile Setup should be set as in the following example:

```
ConnectString='DSN=POLite;DataBase=x:\orahome\POLDB\POLITE;UID=OOT_SCH;PWD=x;
```

## Getting More Information

The only printed information included with the Personal Oracle Lite CD is the *Installation Guide* (CD insert)). All other product-specific information is available online, in printable format. The following table shows how to display this information:

| **For information on...** | **Do this...** |
|---|---|
| SQL Syntax | Double-click the Oracle Lite SQL Language Help icon in the Personal Oracle Lite program group. |
| Replication | Double-click the Oracle Lite Replication Guide icon in the Personal Oracle Lite program group. |
| Oracle Mobile Agents | Refer to the Oracle Mobile Agents user's guide (OMAUSR.PDF) and administrator's guide (OMAADM.PDF) in the \OMA directory on the Personal Oracle Lite CD. |
| **Navigator** | **Click Help or press F1 from any Navigator dialog box.** |
| Data Manager | Click Help or press F1 from any Data Manager dialog box. |
| **SQL*Net** | **Double-click the SQL*Net help icon in the Oracle for Windows 95 or Oracle for Windows NT program group.** |
| SQL*Plus | Double-click the SQL*Plus help icon in the Oracle for Windows 95, Oracle for Windows NT, or Oracle for Windows program group. |
| Late-breaking information about Personal Oracle Lite | Double-click the Release Notes icon in the Personal Oracle Lite program group. |
| Customer Support | Choose Customer Support from the Navigator Help menu. |

**Note:** Items in grey boxes are *only* available with Personal Oracle Lite for Windows 95 and NT.

CHAPTER

# 2 SQL Overview

This chapter provides an overview of SQL and how it is used to organize and manipulate data. Specific topics discussed are:

- SQL fundamentals
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Issuing SQL statements from within a program
- Session and Transaction statements

If you are already familiar with SQL and RDBMS concepts, you may only need to browse through this chapter.

**Note:** This chapter does not attempt to teach new users relational database theory or provide a comprehensive guide to SQL. Many outside sources provide detailed information about RDBMS, such as C.J. Date's *An Introduction to Database Systems* (Addison-Wesley, Reading, Massachusetts, 1983).

## SQL Fundamentals

What we now call SQL is actually a descendant of SEQUEL (or Structured English QUEry Language), which was designed at IBM over twenty years ago. The first commercial RDBMS using SQL appeared in 1981. SQL is now the standard for RDBMS queries across different hardware and software platforms.

SQL is actually a *sublanguage* designed to be embedded in an application programming language. SQL is so flexible that it can be used for:

- data manipulation (retrieval and modification)
- data definition (creating tables and describing the data in each column)
- database administration

SQL has many advantages as a tool for managing relational databases. The two most important are:

- It is supported by many vendors.

  SQL programs are available for almost any server machine in your network. The only requirement is that the communications equipment on either end must use the same SQL dialect.

- It is easy to learn.

  Although SQL is a sophisticated application development language, it uses simple English verbs (such as SELECT, INSERT, UPDATE, DELETE, COMMIT, and ROLLBACK) to describe a desired operation.

SQL database servers handle requests in logical units of work called *transactions*. A transaction is a group of related operations that must all be performed successfully before the RDBMS finalizes any changes to the database.

In SQL, all transactions can be explicitly ended with a command to either accept or discard the changes. Once you are satisfied that no errors occurred during the transaction, you can end that transaction with a COMMIT command. The database then changes to reflect the operations you have just performed. If an error occurs, you can abandon the changes with the ROLLBACK command.

SQL data is manipulated by SQL statements that address the data through a table or view. A view is a virtual table that is defined as a SQL SELECT statement and is not “populated” until it is queried.

### SQL Tables

A database can be made up of one or more database files, or “catalogs” in ODBC and SQL-92. The fundamental unit of storage in SQL is a table consisting of rows of data organized in columns. Each column stores a specific type of data, such as a number or a string of characters. The columns that make up a row contain related data (hence, “relational”), such as an employee name, hire date and salary. This data can then be related to and indexed by an employee number.

All database objects (tables, views or indexes), are owned by a username or a schema. By default, tables are created as part of the “OOT_SCH” schema, the owner of the system tables. The system tables comprise the data dictionary or table of tables.

SQL tables can be quite large, holding thousands or millions of records. To make it easier to navigate through the database tables, users can create an *index* to help access table data more quickly. Just as a book index speeds up the search for a topic in a book, an index created on one or more table columns can make subsequent searches faster. (A drawback of indexes, however, is that they also make subsequent updates to the table slower, since the index adds a level of bookkeeping to database maintenance.)

## Data Definition Language

This section describes SQL commands used to create new database objects, such as schemas, tables, columns, views and sequences.

For information on SQL commands used to query and manipulate data in existing schema objects, see the section “Data Manipulation Language.”

### Naming Database Objects

Object names in SQL must begin with a letter and may contain numbers and the special characters '_' and '$'. Names are generally not case-sensitive, although when enclosed in double quotes (“ ”) mixed case names are permitted.

Object names may be qualified by the catalog and schema to which they belong by separating the qualifiers with a period '`.`' . For example:
`production.payroll.emp.salary` may refer to the `salary` column of the `emp` table owned by the payroll schema in the `production` catalog.

## ALTER SEQUENCE

***Description:*** To change the sequence in one of these ways:

- changing the increment between future sequence values
- setting or eliminating the minimum or maximum value

***Syntax:***

```
ALTER SEQUENCE [<schema_name>.] <sequence_name>
{INCREMENT BY integer
 | {MAXVALUE integer | NOMAXVALUE}
 | {MINVALUE integer | NOMINVALUE}} ...
```

***Key words and parameters:***

*<schema_name>*

is the schema to contain the sequence. If you omit the schema, Personal Oracle Lite alters the sequence in your own schema.

*<sequence_name>*

is the name of the sequence to be altered.

`INCREMENT BY`

specifies the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 9 or fewer digits. The absolute of this value must be less than the difference of `MAXVALUE` and `MINVALUE`. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the interval defaults to 1.

`MAXVALUE`

Specifies the maximum value the sequence can generate. This integer value can have 10 or less digits. MAXVALUE must be greater than MINVALUE.

`NOMAXVALUE`

specifies a maximum value of 2147483647 for an ascending sequence or –1 for a descending sequence.

`MINVALUE`

Specifies the sequence's minimum value. This integer value can have 10 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

`NOMINVALUE`

specifies a minimum value of 1 for an ascending sequence or –2147483647 for a descending sequence.

***Comments:*** The sequence must be dropped and recreated to restart the sequence at a different number. Only future sequence numbers are affected by the ALTER SEQUENCE command.

Some validations are performed. For example, a new MAXVALUE cannot be imposed that is less than the current sequence number.

***Examples:*** This statement sets a new maximum value for the ESEQ sequence:

```
ALTER SEQUENCE eseq
        MAXVALUE 1500
```

## ALTER TABLE

***Description:*** To alter the definition of a table.

***Syntax:***

```
ALTER TABLE [<schema_name>.] <table_name>
ADD ({[COLUMN] <column> <datatype> [DEFAULT expr] [column_constraint]
      [, <column> <datatype> [DEFAULT expr] [column_constraint] ...]
      |table_constraint})
ALTER TABLE [<schema_name>.] <table_name>
DROP {[COLUMN] <column> [RESTRICT|CASCADE]
|<drop_clause>}
```

***Key words and parameters:***

*<schema_name>*
the schema containing the table. If you omit the schema, Personal Oracle Lite assumes the table is in your own schema.

*<table_name>*
the name of the database table.

ADD
adds a column or integrity constraint.

DROP
drops a column or integrity constraint.

COLUMN
specifies a database column.

*<column>*
the name of a database column.

*<datatype>*
the datatype of the column.

DEFAULT
a default value expression for a new or existing column.

*<expr>*
a valid expression.

*<column_constraint>*
a column integrity constraint.

*<table_constraint>*
a table integrity constraint.

RESTRICT

specifies that if any views have definitions that depend on the column to be dropped. the column is not dropped.

CASCADE

specifies that all dependent views are automatically dropped (recursively) with the column.

*<drop_clause>*

an integrity constraint to be dropped.

***Examples:*** The following statement adds a column named THRIFTPLAN of datatype NUMBER with a maximum of seven digits and two decimal places and a column named LOANCODE of datatype CHAR with a size of one and a NOT NULL integrity constraint:

```
ALTER TABLE emp
    ADD (thriftplan NUMBER (7,2),
         loancode CHAR(1))
```

## ALTER VIEW

***Description:*** To recompile a view.

***Syntax:*** ALTER VIEW [*<schema_name>.*] *<view_name>* COMPILE

***Key words and parameters:***

*<schema_name>*

the schema containing the view. If you omit the schema, Personal Oracle Lite assumes the view is in your own schema.

*<view_name>*

the name of the view to be recompiled.

COMPILE

causes Personal Oracle Lite to recompile the view. The COMPILE keyword is required.

***Comments:*** You can use ALTER VIEW to explicitly recompile a view that is invalid. Explicit recompilation allows you to locate recompilation errors before runtime. You may want to explicitly recompile a view after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it. When you issue an ALTER VIEW statement, Personal Oracle Lite recompiles the view regardless of whether it is valid or invalid. Personal Oracle Lite also invalidates any local objects that depend on the view.

**Note:** This command does not change the definition of an existing view. To redefine a view, you must use the CREATE VIEW command with the OR REPLACE option.

***Examples:*** To recompile the view CUSTOMER_VIEW, issue the following statement:

```
ALTER VIEW customer_view
        COMPILE
```

## CREATE DATABASE

***Description:*** Creates a database.

***Syntax:***

```
CREATE DATABASE <database_name> [DATABASE_ID <database_id>]
                                [DATABASE_SIZE <maxbytes>]
                                [EXTENT_SIZE <npages>]
```

***Key words and parameters:***

*<database_name>*

a datafile name or full pathname. Full pathnames must be enclosed in double quotation marks. If no pathname is specified, the datafile will be created in the directory specified by the DSN if connected through ODBC. If neither the full pathname nor DSN are valid, the database will be created under the current working directory. The length of *<database_name>* is limited by the operating system or file system. If a duplicate database name is used, an error is raised (error code -5151, catalog already defined).

*<database_id>*

a unique identifier for the database. Must be a unique number from 0 to 32767. If there is no DSN named `POLITE` in the ODBC.INI file, the `POLITE` DSN will be created. If omitted, the default initial value is 64. The DatabaseID under the `POLITE` DSN indicates the next available database ID. If a database file with the same *<database_id>* already exists, an error is raised (error code -3253, file already exists).

*<maxbytes>*

the maximum file size to which the database can grow. The abbreviations `K`, `M`, and `G` may be used for kilobytes, megabytes, and gigabytes, respectively. If an abbreviation is not specified, the default is `K`. If specifying an abbreviation, you must use an integer value, e.g. `256M`, `1000K`, or `2G`. If omitted, the default value is 256M.

*<npages>*

the number of 4Kbyte pages which will make up an extent (the minimum unit of allocation for a table). A number that is a multiple of 2 is recommended for*<npages>*. If omitted, the default is 4 pages.

***Comments:*** Key words may be listed in any order.

***Examples:*** To create the datafile `LIN.ODB` in the directory `C:\TMP` with the `.ODB` file extension, use:

```
CREATE DATABASE "C:\TMP\LIN"
```

## CREATE INDEX

***Description:*** Creates an index.

***Syntax:***

```
CREATE [ UNIQUE ] INDEX <index_name> ON
  <table_name> ( <index_column_name_list> )
```

***Key words and parameters:***

*<index_name>*

an index name. If a schema name is specified, the index is created under this schema; otherwise, the index is created under the default schema. Any number of indexes can be created for a table, provided that the combination of columns differs for each index.

*<table_name>*

the table on which to build the index. For details on the format of a table name, see CREATE TABLE.

*<index_column_name_list>*

a list of the names of the columns on which to build the index, to a maximum of 16 columns.

***Comments:*** If the optional key word UNIQUE is used, it means the system enforces a UNIQUE constraint on the columns specified by *<index_column_name_list>.*

***Examples:*** To create an index on the HOTEL_NAME column of the HOTEL_DIR table, use:

```
CREATE INDEX HOTEL_INDEX ON HOTEL_DIR(HOTEL_NAME)
```

## CREATE SCHEMA

***Description:*** Creates a SQL schema

***Syntax:***

```
CREATE SCHEMA <schema_name>
```

***Key words and parameters:***

*<schema_name>*

the name of the schema to be created. It is a character string up to 128 characters. The schema name can be prefixed by an optional catalog name. If so, the catalog name and the schema name should be separated by a period '.'. The schema will be created in the specified catalog. If no catalog name is given, the schema will be created in the current default catalog.

***Comments:*** The schema name is usually a user's authorization ID, i.e., USERNAME or a schema name such as SCOTT or PAYABLES. In this case, the schema itself is generally treated as the user's private database. Informally, a schema defines a separate name space and a scope of ownership. In other words, two tables may have the same name if they reside in different schemas. All tables and views in the same schema are owned by the owner of that schema. To use a schema different from the one currently in use, you must first

disconnect from the current schema, then connect to the new schema.

***Examples:*** To create a sample schema called HOTEL_OPERATION, use:

```
CREATE SCHEMA HOTEL_OPERATION
```

To create the schema HOTEL_OPERATION together with the table HOTEL_DIR and the view LARGE_HOTEL use:

```
CREATE SCHEMA HOTEL_OPERATION
  CREATE TABLE HOTEL_DIR(
    HOTELNAME CHAR(40) NOT NULL,
    RATING INTEGER,
    ROOMRATE FLOAT,
    LOCATION CHAR(20) NOT NULL,
    CAPACITY INTEGER)
  CREATE VIEW LARGE_HOTEL AS
    SELECT * FROM HOTEL_DIR
      WHERE CAPACITY > 300
```

## CREATE SEQUENCE

***Description:*** To create a sequence. A sequence is a database object from which multiple users may generate unique integers. Use sequences to automatically generate primary key values.

***Syntax:***

```
CREATE SEQUENCE [<schema_name>.] <sequence_name>
[INCREMENT BY integer]
[START WITH integer]
[{MAXVALUE integer | NOMAXVALUE}]
[{MINVALUE integer | NOMINVALUE}] ...
```

***Key words and parameters:***

*<schema_name>*

is the schema to contain the sequence. If you omit the schema, Personal Oracle Lite creates the sequence in your own schema.

*<sequence_name>*

is the name of the sequence to be created.

INCREMENT BY

specifies the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 10 or fewer digits. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the interval defaults

to 1.

`START WITH`

Specifies the first sequence number to be generated. You can use this option to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the sequence's minimum value or 1. For descending sequences, the default value is the sequence's maximum value or -1. This integer value can have 10 or fewer digits.

`MAXVALUE`

Specifies the maximum value the sequence can generate. This integer value can have 10 or less digits. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.

`NOMAXVALUE`

specifies a maximum value of 2147483647 for an ascending sequence or –1 for a descending sequence.

`MINVALUE`

Specifies the sequence's minimum value. This integer value can have 10 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

`NOMINVALUE`

specifies a minimum value of 1 for an ascending sequence or –2147483647 for a descending sequence.

***Examples:*** The following statement creates the sequence ESEQ:

```
CREATE SEQUENCE eseq
  INCREMENT BY 10
```

## CREATE SYNONYM

***Description:*** Creates a SQL synonym. A synonym is an alternative name for a table, view, sequence, or another synonym.

***Syntax:***

```
CREATE [PUBLIC] SYNONYM [<schema_name>.] <synonym_name>
FOR [<schema_name>.] <object_name>
```

***Key words and parameters:***

`PUBLIC`

creates a public synonym. Public synonyms are accessible to all users. If you omit this option, the synonym is private and is accessible only within its schema.

*`<schema_name>`*

is the schema to contain the synonym. If you omit schema, Personal Oracle Lite creates the synonym in your own schema. You cannot specify schema if you have specified PUBLIC.

*`<synonym_name>`*

is the name of the synonym to be created.

`FOR <object_name>`

identifies the object for which the synonym is created. If you do not qualify the object with a schema, Personal Oracle Lite assumes that the object is in your own schema. The object can be a table, view, sequence, or synonym. Note that the object need not currently exist and you need not have privileges to access the object.

***Comments:*** A private synonym name must be distinct from all other objects in its schema.

***Examples:*** To define the synonym `MARKET` for the table `MARKET_RESEARCH` in the schema `SCOTT`, issue the following statement:

```
CREATE SYNONYM market
  FOR scott.market_research
```

## CREATE TABLE

***Description:*** Creates a SQL table.

***Syntax:***

```
CREATE TABLE <table_name> ( <table_element_list> )
<table_element_list> ::=
    <table_element>
  | <table_element_list>, <table_element>
<table_element> ::=
    <column_definition>
  | <table_constraint_definition>
<column_definition> ::=
    <column_identifier> <data_type> [ <default_value> ]
    [ <column_constraint_definition_list> ]
<default_value> ::=
    DEFAULT <literal>
  | DEFAULT NULL
  | DEFAULT USER
<column_constraint_definition_list> ::=
    <column constraint>
  | <column_constraint_list>
    <column constraint>
<column_constraint> ::=
    NOT NULL
  | UNIQUE
```

```
  | PRIMARY KEY
  | REFERENCES <ref_table_name> <ref_column> [ON DELETE CASCADE]
  | CHECK ( <search_condition> )
<table_constraint> ::=
    UNIQUE ( <column_identifier> )
  | PRIMARY KEY ( <column_identifier> )
  | CHECK ( <search_condition> )
  | FOREIGN KEY <referencing_columns> REFERENCES
    <ref_table_name> <ref_column> [ON DELETE CASCADE]
<column_identifier_list> ::=
    <column_identifier>
  | <column_identifier_list>, <column_identifier>
```

***Key words and parameters:***

*<table_name>*

the name of the table, in the format `catalog_name.schema_name.table_name`.

The catalog name (maximum length: 8 characters) and schema name (maximum length: 128 characters) are optional; if omitted, the default catalog name and schema name are assumed.

Table names may not contain the period "." character, nor begin with an underscore "_" character.

*<column_identifier>*

a column name

*<column_definition>*

the definition of a column. It includes a column name, a data type, an optional default value, and zero or more column constraints.

*<default_value>*

the default value of the column. It can be one of the following:

a. DEFAULT NULL

b. DEFAULT USER (that is, the username when the table is created)

c. a literal

*<column_constraint>*

There are 5 types of column constraints: UNIQUE, PRIMARY KEY, REFERENCES, NOT NULL and CHECK.

- UNIQUE -- a unique constraint. If it appears in *<column_constraint_definition>*, no two rows can have the same value in this column, but multiple NULL values are allowed in this column. If it appears in *<table_constraint_definition>*, no two rows can have identical values for any of the specified columns, but multiple NULL values are allowed in the specified columns.

**Note:** In the current release only one column may be named in each UNIQUE table constraint.

- PRIMARY KEY -- a primary key constraint. If it appears in *<column_constraint_definition>*, no rows in the table can have NULL value in this column, and no two rows may have the same value in this column. If it appears in *<table_constraint_definition>*, none of the columns specified in the constraint can have a NULL value, and no two rows can have identical values for any of the specified columns.
- REFERENCES -- specifies that each value appearing in this column must match one and exactly one value in the referenced column of the referenced table.

  ON DELETE CASCADE -- specifies that Personal Oracle Lite maintains referential integrity by automatically removing dependent foreign key values if you remove a referenced primary or unique key value.
- NOT NULL -- By default, a column may take on NULL values. If you wish to prevent NULL entries, add the NOT NULL constraint.
- CHECK -- specifies a search condition that must be TRUE at all times. Column CHECK constraints and table CHECK constraints are semantically the same.

**Note:** CHECK constraints are not enforced in the current version of Personal Oracle Lite.

*<table_constraint>*

There are four types of table constraints (as described above):

- UNIQUE
- PRIMARY KEY
- CHECK
- FOREIGN KEY -- specifies that each value appearing in this column must match one and exactly one value in the referenced column of the referenced table.

*<ref_table_name>*

the name of the referenced table in a FOREIGN KEY constraint.

*<ref_column>*

the name of the referenced column in a column FOREIGN KEY constraint. If omitted, the referenced column is the PRIMARY KEY of the referenced table.

*<referencing_columns>*

the names of the referencing columns in a table FOREIGN KEY constraint.

*<search_condition>*

specifies a Boolean expression that has the value TRUE, FALSE, or UNKNOWN. For more information, see Search Condition in Chapter 4.

***Comments:*** Each table can have up to 255 columns. Each table can have no more than one primary key constraint. On 16-bit Windows a table can have up to 200 columns.

***Examples:*** The following statement creates a table HOTEL_DIR with two columns: HOTEL_NAME, the primary key, and CAPACITY, which is not nullable and has the default value 0.

```
CREATE TABLE HOTEL_DIR (
  HOTEL_NAME CHAR(40) PRIMARY KEY,
  CAPACITY INTEGER DEFAULT 0 NOT NULL)
```

The following statement creates a table HOTEL_RESTAURANT with three columns:

- REST_NAME is the name of the restaurant
- HOTEL_NAME is the name of the hotel the restaurant is in
- RATING is the rating of the restaurant and its default value is NULL.

The table has the following integrity constraints:

- no two hotel restaurants have the same name
- HOTEL_NAME must refer to a hotel in the HOTEL_DIR table

```
CREATE TABLE HOTEL_RESTAURANT(
  REST_NAME CHAR(50) UNIQUE,
  HOTEL_NAME CHAR(40)  REFERENCES HOTEL_DIR,
  RATING FLOAT DEFAULT NULL)
```

## CREATE VIEW

***Description:*** Creates a view.

***Syntax:***

*CREATE [or REPLACE]* VIEW *<view_name>*

*[ <column_comma_list> ]* AS *<query_spec>*

***Key words and parameters:***

*[or REPLACE]*

an optional key word. If used, it means if there is a view with the same name, the original view is replaced by the new definition.

*<view_name>*

a full view name consists of a schema name and a table name. If the schema name is omitted, the default schema name is assumed.

*<column_comma_list>*

a list of column names in the view table. The number of names must match that of the select list specified in *<query_spec>*. If omitted, the column names are the

column names of the selected expression in *<query_spec>*.

*<query_spec>*

the definition of how the view is derived. It is basically a select statement on one or more tables.

***Comments:*** A view is updatable if:

- *<query_spec>* selects from a single base table or from another updatable view
- each selected expression is a column reference to that base table or updatable view
- no two column references in the select list reference the same column

***Examples:*** The following statement creates a view called HOTEL_INFO that contains the hotel and restaurant information.

HOTEL_INFO has five columns.

- HOTEL_NAME is derived from the HOTEL_NAME column of HOTEL_DIR table.
- HOTEL_RATING is derived from the RATING column of HOTEL_DIR.
- HOTEL_CAPACITY is derived from the CAPACITY column of the HOTEL_DIR table.
- REST_NAME is derived from the REST_NAME column of the HOTEL_RESTAURANT table.
- REST_RATING is derived from the RATING column of the HOTEL_RESTAURANT table.

```
CREATE VIEW HOTEL_INFO (HOTEL_NAME, HOTEL_RATING,
    HOTEL_CAPACITY, REST_NAME, REST_RATING) AS
  SELECT a.HOTEL_NAME, a.RATING,
    a.CAPACITY, b.REST_NAME, b.REST_RATING
  FROM HOTEL_DIR a, HOTEL_RESTAURANT b
  WHERE a.HOTEL_NAME = b.HOTEL_NAME
```

## DROP INDEX

***Description:*** Drops an index on a table.

***Syntax:*** `DROP INDEX <index_name>`

***Key words and parameters:***

*<index_name>*

the name of the index to drop. If the schema name is not specified, the default schema name is assumed.

***Examples:*** To drop the index named HOTEL_INDEX in the HOTEL_OPERATION schema of the current catalog, use:

```
DROP INDEX HOTEL_OPERATION.HOTEL_INDEX;
```

## DROP SCHEMA

***Description:*** Drops a schema from a database.

***Syntax:*** `DROP SCHEMA <schema_name> [ <drop_behavior> ]`

***Key words and parameters:***

*<schema_name>*

an optional catalog name and a schema name. If catalog name is omitted, the current default catalog is assumed.

*<drop_behavior>*

a specification of drop behavior, which is either the key word RESTRICT or CASCADE. RESTRICT is assumed if it is not specified. RESTRICT means the DROP SCHEMA statement will fail if there exist any dependent schema objects, such as tables, views, or indexes under the schema. CASCADE means all dependent schema objects, such as tables, views, and indexes will be automatically dropped along with the schema.

***Examples:*** To drop the schema called HOTEL_OPERATION in the default catalog, use:

```
DROP SCHEMA HOTEL_OPERATION
```

## DROP SEQUENCE

***Description:*** Drops a sequence from the database.

***Syntax:*** `DROP SEQUENCE [<schema_name>.] <sequence_name>`

***Key words and parameters:***

*<schema_name>*

is the schema containing the sequence. If you omit the schema, Personal Oracle Lite drops the sequence in your own schema.

*<sequence_name>*

is the name of the sequence.

***Examples:*** The following statement drops the sequence ESEQ:

```
DROP SEQUENCE eseq
```

## DROP SYNONYM

***Description:*** Drops a synonym from a database.

***Syntax:***

```
DROP [PUBLIC] SYNONYM [<schema_name>.] <synonym_name>
```

***Key words and parameters:***

`PUBLIC`

must be specified to drop a public synonym. You cannot specify *<schema_name>* if you have specified `PUBLIC`.

*<schema_name>*

is the schema containing the synonym. If you omit the schema, Personal Oracle Lite assumes the synonym is in your own schema.

*<synonym_name>*

is the name of the synonym to be dropped

***Examples:*** To drop a synonym named `MARKET`, issue the following statement:

```
DROP SYNONYM market
```

## DROP TABLE

***Description:*** Drops a table, including both definition and data.

***Syntax:***

```
DROP TABLE [<schema_name>.] <table_name>
   [CASCADE | CASCADE CONSTRAINTS | RESTRICT]
```

***Key words and parameters:***

*<schema_name>*

the schema containing the table. If you omit the schema, Personal Oracle Lite assumes the table is in your own schema.

*<table_name>*

the name of the database table.

`CASCADE`

specifies that, if the table is a base table for views, or if there are referential integrity constraints that refer to primary keys in the table, they are automatically dropped with the table.

`CASCADE CONSTRAINTS`

specifies that all referential integrity constraints that refer to primary keys in the table are automatically dropped with the table.

`RESTRICT`

specifies that, if the table is a base table for views, or if the table is referenced in any referential integrity constraints, the DROP TABLE operation fails.

***Examples:*** To drop the table `HOTEL_DIR`, use:

```
DROP TABLE HOTEL_DIR RESTRICT
```

### DROP VIEW

***Description:*** Drops the definition of a view.

***Syntax:***

```
DROP VIEW <view_name> [ <drop_behavior> ]
```

***Key words and parameters:***

*<view_name>*

the name of the view to drop. If the schema name is not specified, the current default schema name is assumed.

*<drop_behavior>*

a specification of the drop behavior, which is either RESTRICT or CASCADE. RESTRICT is assumed if it is not specified. RESTRICT means if any views exist whose definition depend on the view, the DROP VIEW statement fails. CASCADE means all dependent views are automatically dropped (recursively) along with the view.

***Examples:*** To drop the view `BEST_HOTELS`, use:

```
DROP VIEW BEST_HOTELS RESTRICT
```

## Data Manipulation Language

This section describes SQL commands used to query and manipulate data in existing schema objects.

Some of the examples in this section are based on the table, `PARTS`, with three columns: `ID`, `NAME`, and `WEIGHT`. The table is populated as follows:

| ID | NAME | WEIGHT |
|---|---|---|
| P1 | Nut | 13.00 |
| P2 | Bolt | 18.00 |
| P3 | Screw | 16.00 |
| P4 | Screw | 12.00 |
| P5 | Cog | 20.00 |
| P6 | Cam | 14.00 |

For information on SQL commands used to create new schema objects, see the section “Data Definition Language.”

## DELETE

***Description:*** Deletes rows from a table.

***Syntax:***

```
DELETE FROM <table_name>
DELETE FROM <table_name> WHERE <search_condition>
DELETE FROM <table_name> WHERE CURRENT OF <cursor_name>
```

***Key words and parameters:***

*<table_name>*
the name of a base table.

*<search_condition>*
see Search Condition.

*<cursor_name>*
name of a cursor (an identifier).

***Comments:*** If no WHERE clause is specified, then all rows of *<table_name>* are deleted.

A positioned delete requires that the cursor be updatable.

***Examples:***

```
DELETE * FROM PARTS
DELETE FROM PARTS WHERE NAME = 'Screw'
DELETE FROM PARTS WHERE CURRENT OF DiscontinuedParts
```

## INSERT

***Description:*** Inserts a new row into a table (or view).

***Syntax:***

```
INSERT INTO <table_name> DEFAULT VALUES
INSERT INTO <table_name> [ <column_commalist> ]
  VALUES ( <insert_value_list> )
INSERT INTO <table_name> [ <column_commalist> ] ( <query_spec> )
```

***Key words and parameters:***

*<table_name>*
name of a base table.

*<column_commalist>*
a list of column names where the values specified in *<insert_value_list>* are filled. The remaining columns are filled with default values. If the *<column_commalist>* is not provided, then the values specified in *<insert_value_list>* are filled in from left to right in the table, starting from

the first column.

*<insert_value_list>*

a list of insert values, separated by commas. An insert value is one of the following: a value expression, NULL, or DEFAULT.

*<query_spec>*

a subquery specification.

***Comments:*** The first form of INSERT statement fills each column with its default value, as defined when the table was created. The second form of INSERT statement fills each column with values listed in *<insert_value_list>*.

The same column name may not appear more than once in the *<column_commalist>*.

The number of column names specified in *<column_commalist>* must be the same as the number of values provided in *<insert_value_list>*.

If *<column_commalist>* is omitted, then the number of values in *<insert_value_list>* must be equal to the degree of *<table_name>*.

It is recommended that the *<column_commalist>* is always used such that the INSERT statement, in most cases, still remains valid even after the table definition is altered.

If a column does not have a user-defined default value, its default value is NULL. This is true even though there exists a NOT NULL constraint on the column. Therefore, if an INSERT statement does not provide an explicit value for such a column, an integrity violation error message is generated.

***Examples:*** The following INSERT statement adds a new part (P7,'Nail',10) to table PARTS.

```
INSERT INTO PARTS VALUES ('P7', 'Nail', 10)
```

The following INSERT statement is similar to the previous INSERT statement except it lists all column names explicitly in an order different from their creation order. In addition, it converts the weight from kilogram to pound.

```
INSERT INTO PARTS (NAME, ID, WEIGHT)
  VALUES ('Nail', 'P7', 4.54 / 0.454)
```

## SELECT

***Description:*** Retrieves data from a table.

***Syntax:***

```
SELECT [DISTINCT | ALL] { *
                        | { [<schema_name>.]{<table_name> | <view_name>}.*
```

```
                    | expr [c_alias] }
                   [, { [<schema_name>.]{<table_name> | <view_name>}.*
                    | expr [c_alias] } ] ... }
  FROM [<schema_name>.]{<table_name> | <view_name>} [t_alias]
    [, [<schema_name>.]{<table_name> | <view_name>} [t_alias] ] ...
  [WHERE condition ]
  [CONNECT BY condition }
  [GROUP BY expr [, expr] ... [HAVING condition] ]
  [{UNION | UNION ALL |MINUS} SELECT command ]
  [ORDER BY {expr | position} [ASC | DESC]
         [, {expr | position} [ASC | DESC]] ...]
```

***Key words and parameters:***

DISTINCT

returns only one copy of each set of duplicate rows selected. Duplicate rows are those with matching values for each expression in the select list.

ALL

returns all rows selected, including all copies of duplicates. The default is ALL.

*<*>*

selects all columns from all tables, views, or snapshots listed in the FROM clause.

*<table_name.*>*

selects all columns from the selected table. Use the *schema* qualifier to select from a schema other than your own.

*<view_name.*>*

selects all columns from the selected view. Use the *schema* qualifier to select from a schema other than your own.

*<expr>*

selects an expression, usually based on column values, from one of the tables, views, or snapshots in the FROM clause. A column name in this list can be qualified with *schema* only if the table, view, or snapshot that contains the column is itself qualified with *schema* in the FROM clause.

For a description of valid Personal Oracle Lite expressions, see Chapter 4.

*<c_alias>*

provides a column alias, which is a different name for the column expression, and causes the column alias to be used in the column heading. A column alias does not affect the actual name of the column. Once aliased, you must use the alias for the remainder of the query.

*<schema_name>*

the schema that contains the selected table, view, or snapshot. If you omit *schema*, Personal Oracle Lite assumes that the table, view, or snapshot resides in your own schema.

*<table_name>*

the table from which data is selected.

*<view_name>*

the view from which data is selected.

<*t_alias*>

provides a different name, or alias, for the table, view, or snapshot for the purpose of evaluating the query. Most often used in a correlated query. Other references to the table, view, or snapshot throughout the query must refer to the alias.

WHERE

restricts the rows selected to those for which the specified *condition* is TRUE. If you omit the WHERE clause, Personal Oracle Lite returns all rows from the tables, views, or snapshots in the FROM clause.

In an embedded SQL SELECT statement, the *condition* in a WHERE clause can contain host variables. For more information about creating a valid *condition*, refer to Chapter 5.

START WITH

specifies the root rows of the hierarchy in a hierarchical query.

CONNECT BY

specifies the relationship between parent and child rows in a hierarchical query. The *condition* defines this relationship, and must use the PRIOR operator to refer to the parent row. To find the children of the parent row, Personal Oracle Lite evaluates the PRIOR expression for each row in the table. Rows for which the condition is TRUE are the children of the parent.

GROUP BY

groups the selected rows based on the value of *<expr>* for each row, and returns a single row of summary information for each group.

HAVING

restricts the groups of rows returned to those groups for which *condition* is TRUE. If you omit this clause, Personal Oracle Lite returns summary rows for all groups. For more information about creating a valid *condition*, refer to Chapter 5.

UNION

returns all distinct rows selected by either query.

UNION ALL

returns all rows selected by either query, including duplicates.

MINUS

returns all distinct rows selected by the first query but not the second.

ORDER BY

orders rows returned by the SELECT statement, according to the following arguments:

*expr* orders rows based on their value for *<expr>*. The expression is based on columns in the select list, or based on columns in the tables, views, or snapshots in the FROM clause.

*position* orders rows based on their value for the expression in this position in the select list. For example, ORDER BY 1 would sort by the first expression in the select list.

ASC specifies an ascending sort order. ASC is the default.

DESC specifies a descending sort order.

***Comments:*** If no WHERE clause is specified and there is more than one table in the FROM clause, the database system computes a Cartesian product of all the tables involved.

All columns named in GROUP BY or ORDER BY clauses must be in the select list. If you give a column an alias in the select list, you must refer to it by the alias, not the column name, for the remainder of the statement.

A SELECT statement that performs a hierarchical query can use the LEVEL pseudocolumn. LEVEL returns the value 1 for a root node, 2 for a child node, and 3 for a grandchild, etc. For more information on LEVEL, see Chapter 3.

A hierarchical query cannot also perform a join, nor can it select data from a view.

***Examples:*** The following SELECT statement retrieves all parts with weight greater than or equal to 16:

```
SELECT * FROM PARTS WHERE WEIGHT >= 16
```

The result is:

```
P2 Bolt 18.00
P3 Screw 16.00
P5 Cog 20.00
```

The following SELECT statement is similar to the previous statement, except that it displays the resulting rows in an ascending order on WEIGHT and the strings to annotate the meaning of the retrieved columns:

```
SELECT 'ID=',ID, ', Name=',NAME, ', Weight=',WEIGHT
  FROM PARTS WHERE WEIGHT >= 16 ORDER BY WEIGHT ASC
```

The result is:

```
ID=P3, Name=Screw, Weight=16.00
ID=P2, Name=Bolt, Weight=18.00
ID=P5, Name=Cog, Weight=20.00
```

The following SELECT statement converts pound to kilogram for WEIGHT and displays the resulting rows in a descending order of the converted weight.

```
SELECT 'ID=', ID, ', Name=', NAME, ', Weight(kg)=',
```

```
WEIGHT * 0.454 AS WEIGHT_KG
FROM PARTS WHERE WEIGHT >= 16 ORDER BY WEIGHT_KG DESC
```

The result is:

```
ID=P5, Name=Cog, Weight(kg)=9.08
ID=P2, Name=Bolt, Weight(kg)=8.17
ID=P3, Name=Screw, Weight(kg)=7.26
```

The following SELECT statement lists all pairs of parts in PARTS that have the same NAME value, but different ID value.

```
SELECT * FROM PARTS a, PARTS b
  WHERE a.ID <> b.ID AND a.NAME = b.NAME
```

The result is:

```
P3 Screw 16.00 P4 Screw 12.00
P4 Screw 12.00 P3 Screw 16.00
```

The following CONNECT BY clause defines a hierarchical relationship in which the EMPNO value of the parent row is equal to the MGR value of the child row:

```
CONNET BY PRIOR empno = mgr
```

In the following CONNECT BY clause, the PRIOR operator applies only to the EMPNO value. To evaluate this condition, Personal Oracle Lite evaluates EMPNO values for the parent row and MGR, SAL, and COMM values for the child row:

```
CONNET BY PRIOR empno = mgr AND sal > comm
```

To qualify as a child row, a row must have a MGR value equal to the EMPNO value of the parent row and it must have a SAL value greater than its COMM value.

## Outer Joins

A join is a query that combines rows from two or more tables, views, or snapshots. Personal Oracle Lite performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a join condition. To execute a join, Personal Oracle Lite combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not also appear in the select list.

The outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and those rows from one table for which no rows from the other satisfy the join condition. Such rows are not returned by a simple join. To write a query that performs an outer join of tables A and B and returns all rows from A, apply the outer join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, Personal Oracle Lite returns NULL for any select list expressions containing columns of B. The following is the basic syntax of an outer join of two tables:

```
SELECT [table] .column
       [, [table] .column]...
       FROM table1, table2
       WHERE {table1.column = table2.column (+)
             |table1.column (+) = table2.column}
```

In the following outer join query, Personal Oracle Lite returns a row containing the OPERATIONS department even though no employees work in this department. It returns NULL in the ENAME and JOB columns for this row:

```
SELECT ename, job, dept.deptno, dname
        FROM emp, dept
        WHERE emp.deptno (+) = dept.deptno
```

```
ENAME      JOB           DEPTNO DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
                             40 OPERATIONS
```

## UPDATE

***Description:*** Updates rows of a table.

***Syntax:*** `UPDATE <table_name> SET <set_clause_list>`

```
UPDATE <table_name> SET <set_clause_list>
  WHERE <search_condition>
UPDATE <table_name> SET <set_clause_list>
  WHERE CURRENT OF <cursor_name>
```

***Key words and parameters:***

*<table_name>*
the name of a base table.

*<column_identifier>*
a column name defined in table <table_name> .

*<value_expression>*
see Value Expression.

*<search_condition>*
see Search Condition.

*<cursor_name>*
name of a cursor (an identifier).

***Comments:***

The same column name may not appear more than once in the *<set_clause_list>*.

If no WHERE clause is specified, then all rows of *<table_name>* are updated.

A positioned update requires that the cursor be updatable.

***Examples:***

```
UPDATE PARTS SET WEIGHT = WEIGHT * 0.454
UPDATE PARTS SET WEIGHT = 14 WHERE NAME = 'Screw'
```

## Issuing SQL Statements From Within A Program

You can issue SQL statements in a host language (such as C) in either of two ways:

- Embedding SQL statements within the code

  As noted earlier in this chapter, SQL is designed to be embedded within a program. The embedded SQL statements must be “pre-processed” into constructs and statements of the host language, which is then compiled by a standard compiler for that language.

- Interactively

  If you connect to the database from within the application, using the appropriate ODBC driver, you can pass SQL statements directly to the server. The ODBC driver -- a C library through which an application program can issue SQL statements composed at runtime -- defines a call-level SQL interface from the

host language.

Personal Oracle Lite's implementation of SQL complies with both ANSI SQL-92 and ODBC 2.0. The interoperability of Personal Oracle Lite-supported data types and object classes makes it possible to use SQL statements to access objects created by C/C++ programs and vice versa.

## Session and Transaction Statements

With the exception of the `GET DIAGNOSTIC` and `CONNECT` statements in embedded SQL, the execution of all other SQL statements require the existence of a SQL session. An application can establish a SQL session by:

- issuing a SQL statement that requires a SQL session so that a default session is implicitly established
- issuing `SQLConnect` or `SQLDriverConnect` ODBC calls

A SQL session is closed when one of the following occurs:

- an embedded SQL program terminates
- the `SQLDisconnect` API in ODBC is called
- an ODBC program terminates

**Transactions**

Most Oracle tools make a connection when you specify a connect string on the command line, or you fill in a dialog box with a username, password and a connect string, as in the following example:

ODBC : POlite or

ODBC: <your DataSourceName>

A SQL transaction starts when any DDL or DML statement is executed in a session (which ends when you do a COMMIT or ROLLBACK).

For embedded SQL, opening a cursor starts a transaction (which ends when you do a COMMIT or ROLLBACK). For ODBC, a transaction is terminated by calling SQLTransact.

The execution of all statements in the same transaction is atomic. If a transaction is committed, the effects of executing all statements in the transaction become permanent in the database. If a transaction is rolled back, the database is restored to the original state as if none of these statements had been executed.

If multiple ODBC connections are open to the same Personal Oracle Lite database, a COMMIT (or ROLLBACK) on one connection commits (or rolls back) all transactions from all connections.

## COMMIT

***Description:*** Commits a transaction.

***Syntax:***

```
COMMIT [WORK]
```

***Key words and parameters:*** `WORK` is optional.

***Examples*** To commit the current transaction in an embedded SQL program, use:

```
EXEC SQL COMMIT WORK
```

## ROLLBACK

***Description:*** Rolls back a transaction.

***Syntax:***

```
ROLLBACK [WORK]
```

***Key words and parameters:*** `WORK` is optional.

***Examples:*** To roll back the current transaction, use:

```
ROLLBACK WORK
```

or simply

```
ROLLBACK
```

CHAPTER

# 3 Data Types and Literals

This chapter presents the following topics:

- SQL (Structured Query Language) standards and the Open Database Connectivity Interface (ODBC)
- Data types, literals, and other SQL elements supported by Personal Oracle Lite
- C data types supported by ODBC and Personal Oracle Lite

## Overview

This chapter describes the following:

- SQL (Structured Query Language) standards and the Open Database Connectivity Interface (ODBC).
- Data types, literals, and other SQL elements supported by Personal Oracle Lite.
- C data types supported by ODBC and Personal Oracle Lite.

### SQL Standards

SQLwas first developed at IBM in the mid 1970s and became widely accepted commercially during the 1980s. The first SQL standard was published by ANSI in 1989, commonly known as SQL-89. In 1992, ANSI enhanced the SQL standard by clarifying some of the ambiguities and by including new features like dynamic SQL, descriptors, and diagnostics areas. This new standard is known as SQL-92, or SQL-2. Oracle established its own set of datatypes using many of the same type names, but with slightly different meanings.

### SQL and ODBC

The SQL-92 standard, however, does not address the issue of connecting applications to DBMSs from different vendors. The Open Database Connectivity Interface (ODBC) from Microsoft defines a call level interface to provide interoperability across different DBMSs. ODBC is based on the X/Open and SQL Access Group (SAG) Call Level Interface (CLI). It specifies a set of interface functions to allow:

- connections to databases by different vendors
- preparation and execution of SQL statements in a common language
- retrieval of query results into local program variables

Personal Oracle Lite SQL supports both the SQL-92 embedded SQL C binding and ODBC 2.0 call level interface. Personal Oracle Lite SQL supports implicit type conversion from character string type to another data type whenever necessary. For example, if the data type of a column `AGE` is `INTEGER` and the statement `UPDATE EMPLOYEE SET AGE = '30' WHERE NAME = 'John'` is executed, `'30'` is automatically converted to an `INTEGER` type.

The discussion on SQL data types, literals, expressions, and predicates is generally applicable to both SQL-92 and ODBC CLI. When necessary, specific features are annotated with `[ODBC]` to indicate that the feature is defined by ODBC, and with `[SQL-92]` to indicate that the feature is defined by SQL-92. Personal Oracle Lite

SQL has also added a limited number of Oracle extensions, which are indicated by `[Oracle]`. Type names of the form “SQL_C_Name” refer to the type of variable in a client program which can accept data from the SQL datatype (also known as binding).

## Data Types, Literals, and Other SQL Elements

This section describes data types, literals, and other SQL elements supported by Personal Oracle Lite.

### Data Types

### CHAR

[ODBC] [SQL-92] [Oracle]

***Description:*** Fixed length character string type. CHAR columns allocate a fixed space in a database row, allowing for the maximum length. Strings shorter than the maximum are padded with trailing blanks.

***Syntax:***

```
CHAR
CHARACTER
CHAR ( <length> )
CHARACTER ( <length> )
```

***Key words and parameters:***

*`<length>`* the number of characters in a string, between 1 and 2048. The limit is 255 for Oracle7.

***Comments:*** If *`<length>`* is omitted, 1 is assumed.

***Examples:***

```
CHAR
CHAR(20)
```

### VARCHAR

[ODBC] [SQL-92] [Oracle]

***Description:*** Variable-length character string type.

***Syntax:***

```
VARCHAR ( <length> )
CHAR VARYING ( <length> )
CHARACTER VARYING ( <length> )
```

***Key words and parameters:***

*<length>* the maximum number of characters in a string, between 1 and 2048.

***Comments:*** If *<length>* is omitted, 1 is assumed. For Oracle7, the maximum length is 2000.

***Examples:***

```
VARCHAR(20)
CHAR VARYING(20)
CHARACTER VARYING(20)
```

## VARCHAR2

[Oracle]

***Description:*** Variable-length character string type.VARCHAR and VARCHAR2 are stored exactly as passed, provided the length does not exceed the maximum. No blank padding is added. VARCHAR and VARCHAR2 are equivalent.

***Syntax:***

```
VARCHAR2 ( <length> )
CHAR VARYING ( <length> )
CHARACTER VARYING ( <length> )
```

***Key words and parameters:***

*<length>* the maximum number of characters in a string, between 1 and 2048. (2000 for Oracle7).

***Comments:*** If *<length>* is omitted, 1 is assumed.

***Examples:***

```
VARCHAR2(20)
CHAR VARYING(20)
CHARACTER VARYING(20)
```

## LONG VARCHAR

[ODBC]

***Description:*** Variable-length character string type. Used when the length of the string exceeds 2048 bytes.

***Syntax:***

```
LONG VARCHAR
LONG VARCHAR ( <length> )
```

***Key words and parameters:***

*<length>* the maximum number of characters in a string.

***Comments:*** The maximum length of a `LONG VARCHAR` is 2 billion bytes. If *<length>* is omitted, 2 megabytes is assumed. You can create an index on a LONG VARCHAR column, but only the first 2000 bytes will be used in the index.

***Examples:*** `LONG VARCHAR`

### LONG

[Oracle]

***Description:*** Same as LONG VARCHAR. Contains up to 2 billion bytes of character data.

### TINYINT

[ODBC]

***Description:*** A one byte integer with range 0 to 127 if unsigned (SQL_C_UTINYINT) or - 128 to + 127 if signed (SQL_C_STINYINT).

### SMALLINT

[ODBC] [SQL-92]

***Description:*** Small integer type.

***Syntax:*** `SMALLINT`

***Comments:*** A `SMALLINT` is an exact numeric value with precision 5 and scale 0, typically 2 bytes or 16 bits. If signed, the range can be -32,768 to +32,767 (SQL_C_SSHORT or SQL_C_SHORT) or, if unsigned, 0 to 65,535 (SQL_C_USHORT).

-32768 <= n <= 32767, where n is the value of a `SMALLINT.`

***Examples:*** `SMALLINT`

### INTEGER

[ODBC] [SQL-92]

***Description:*** Integer type.

***Syntax:*** `INTEGER`

`INT`

***Comments:*** An `INTEGER` is an exact numeric value with precision 10 and scale 0, typically 4 bytes. Binds with SQL_C_LONG or SQL_C_ULONG and SQL_C_SLONG.

-2^31 < n < 2^31, where n is the value of an `INTEGER`.

***Examples:*** `INTEGER`

`INT`

### BIGINT

[ODBC]

***Description:*** Big integer type. Binds with SQL_C_CHAR or SQL_C_BINARY variables.

***Syntax:***

```
BIGINT
```

***Comments:*** A `BIGINT` is an exact numeric value with precision 19 and scale 0, typically 8 bytes.

-10^19 < n < 10^19, where n is the value of a `BIGINT`.

***Examples:***

```
BIGINT
```

### DECIMAL

[ODBC] [SQL-92]

***Description:*** Decimal number type.

***Syntax:***

```
DECIMAL [ ( <precision>[, <scale> ] ) ]
DEC [ ( <precision>[, <scale> ] ) ]
```

***Key words and parameters:***

*`<precision>`* the precision of a decimal number.

*`<scale>`* the scale of a decimal number (the number of digits to the right of the decimal point).

***Comments:*** A `DECIMAL` is an exact numeric value. By default, DECIMAL data is returned as a character string or SQL_C_CHAR, but conversion into SQL_C_LONG or SQL_C_FLOAT or other datatypes is supported.

If *`<precision>`* is not specified, 38 is assumed. If *`<scale>`* is not specified, 0 is assumed.

0 <= *`<scale>`* <= *`<precision>`* <= 38.

***Examples:***

```
DECIMAL
DEC (5)
DECIMAL (10, 5)
```

### NUMERIC

[ODBC] [SQL-92}

***Description:*** `NUMERIC` is synonymous with `DECIMAL.`

### NUMBER

[Oracle]

***Description:*** NUMBER is synonymous with DECIMAL and NUMERIC.

### REAL

[ODBC]

***Description:*** Floating point number type. Binds with SQL_C_REAL variables.

***Syntax:***

```
REAL
```

***Comments:*** A `REAL` is a signed approximate numeric value with a mantissa decimal precision 7. Its absolute value is either zero or between 10^-38 and 10^38.

***Examples:***

```
REAL
```

### FLOAT

[ODBC]

***Description:*** Floating point number type. Binds with a SQL_C_DOUBLE variable.

***Syntax:***

```
FLOAT [ ( <precision> ) ]
```

***Key words and parameters:***

*`<precision>`* the precision of a floating point number.

***Comments:*** A `FLOAT` is a signed approximate numeric value with a mantissa decimal precision 15. Its absolute value is either zero or between 10^-308 and 10^308. In the current implementation, the precision of a `FLOAT` is always set to 15.

***Examples:***

```
FLOAT
FLOAT (10)
```

### DOUBLE PRECISION

[ODBC]

***Description:*** Double precision floating point number type. Binds with a SQL_C_DOUBLE variable

***Syntax:***

```
DOUBLE PRECISION
```

***Comments:*** A `DOUBLE PRECISION` is a signed, approximate, numeric value with a mantissa decimal precision 15. Its absolute value is either zero or between 10^-308 and 10^308.

***Examples:***

```
DOUBLE PRECISION
```

### BINARY

[ODBC]

***Description:*** Variable length binary data type. Binds with a SQL_C_CHAR or SQL_C_BINARY array.

***Syntax:*** `BINARY [( <precision> )]`

***Key words and parameters:*** *<precision>* maximum number of bytes.

***Examples:*** `BINARY(1024)`

### VARBINARY

[ODBC]

***Description:*** `VARBINARY` is synonymous with `BINARY`.

### RAW

[Oracle]

***Description:*** `RAW` is synonymous with `BINARY`, but has a limit of 255 bytes in Oracle, 2048 in Personal Oracle Lite. Binds with SQL_C_BINARY or SQL_C_CHAR arrays.

### LONG VARBINARY

[ODBC]

***Description:*** Variable length binary data type.

***Syntax:*** `LONG BINARY [( <precision> )]`

***Key words and parameters:*** `<precision>` maximum number of bytes. If not specified, default is 2 megabytes.

***Comments:*** 1 <= *<precision>* <= 2G.

***Examples:*** `LONG VARBINARY(1048576)`

### LONG RAW

[Oracle]

***Description:*** Variable length binary data type. Similar to LONG VARBINARY. Use this type when a VARBINARY column exceeds 2048 bytes.

***Syntax:*** `LONG RAW [( <precision> )]`

***Key words and parameters:*** `<precision>` maximum number of bytes. If not specified, default is 2 megabytes.

***Examples:***

```
LONG RAW(1048576)
```

### DATE

[ODBC] [SQL-92]

***Description:*** Date type. Stores day, month and year in SQL-92 and ODBC. In Oracle, it also stores the time.

***Syntax:***

```
DATE
```

***Examples:***

```
DATE
```

### TIME

[ODBC] [SQL-92]

***Description:*** Time type. Stores hour, minutes, seconds and possibly, fractional seconds.

***Syntax:***

```
TIME
TIME ( <precision> )      [SQL-92]
```

***Key words and parameters:*** *`<precision>`* the number of fractional digits in seconds.

***Examples:***

```
TIME
TIME (3)
```

### TIMESTAMP

[ODBC] [SQL-92]

***Description:*** Timestamp type. Stores both date and time in SQL-92 and is comparable to Oracle DATE data type.

***Syntax:***

```
TIMESTAMP [ ( <precision> ) ]
```

***Key words and parameters:*** *`<precision>`* the number of fractional digits in seconds. 0 <= `<precision>` <= 6

***Syntax:*** During replication of an Oracle7 table, DATE columns in Oracle7 are stored as TIMESTAMP columns in Personal Oracle Lite.

***Examples:***

```
TIMESTAMP
TIMESTAMP (3)
```

### ROWID

***Description:*** A 16-byte hexadecimal string representing the unique address of a row in its table. `ROWID` is primarily for values returned by the `ROWID` pseudocolumn.

**Note:** In Personal Oracle Lite, the ROWID is the hexadecimal string representing the unique object identifier. It is not compatible with the Oracle ROWID, but it may be used to uniquely identify a row for updating.

ROWID literals should be enclosed in single quotes.

***Syntax:*** none

***Examples:***
```
A80000.00.03000000
```

## Literals

### CHAR, VARCHAR

***Description:*** Character string literal value.

***Syntax:***
```
'<letters>'
```

***Key words and parameters:***

`<letters>` a sequence of zero or more printable characters excluding new-line.

***Comments:*** If a single quote is part of a literal, it must be preceded by another single quote (used as an escape character). The maximum length of a character literal is 1024.

***Examples:***
```
'a string'
'a string containing a quote '''
```

### SMALLINT, INTEGER, BIGINT, TINYINT

[ODBC]

***Description:*** Integer literal value.

***Syntax:***
```
[+|-]<digits>
```

***Key words and parameters:***

*`<digits>`* a sequence of one or more digits.

***Comments:*** Let n be the number the literal represents.

For `TINYINT`, -128 <=n <= 127

For a `SMALLINT`, -32768 <= n <= 32767.

For an `INTEGER`, -2^31 < n < 2^31.

For a `BIGINT`, -10^19 < n < 10^19

***Examples:*** `12345`

### DECIMAL, NUMERIC, NUMBER

***Description:*** Decimal number literal value.

***Syntax:***

*`[+|-]<digits>`*

*`[+|-]<digits>.[<digits>]`*

*`[+|-].<digits>`*

***Key words and parameters:***

*`<digits>`* a sequence of one or more digits.

***Examples:***

`54321`

`-123.`

`+456`

`64591.645`

`+.12345`

`0.12345`

### REAL, FLOAT, DOUBLE PRECISION

***Description:*** Floating point number literal value.

***Syntax:***

*`[+|-]<digits><exp>[+|-]<digits>`*

*`[+|-]<digits>.[<digits>]<exp>[+|-]<digits>`*

*`[+|-].<digits><exp>[+|-]<digits>`*

***Key words and parameters:***

*`<digits>`* a sequence of one or more digits.

*`<exp>`* 'E' or 'e.'

***Examples:***

`+1.5e-7`

`12E-5`

`-.12345e+6789`

### DATE

***Description:*** Date literal value.

***Syntax:***

`[DATE] '`*`<year1><month1><day>`*`'` `[SQL-92]`

`{ d '`*`<year1><month1><day>`*`'` `[ODBC]`

```
--(* d '<year1><month1><day>' *)--    [ODBC]
'<day><month2><year2>'                [Oracle]
'<day><month2><year1>'                [Oracle]
'<month1><day><year2>'                [Oracle]
'<month1><day><year1>'                [Oracle]
```

***Key words and parameters:***

| | |
|---|---|
| <year1> | a four digit number representing a year, e.g., 1994. |
| <year2> | a two digit number representing the last two digits of a year. |
| *<month1>* | a two digit number between 01 and 12. |
| *<month2>* | a three letter initial of a month (case insensitive). |
| *<day>* | a two digit number between 01 and 31 (depending on the month). |

***Examples:***

```
'1994-11-07' [SQL-92]
{ d '1994-11-07' }
--(* d '1994-11-07' *)--
DATE '10-23-94'
'23-Nov-1994' [Oracle]
'23-Nov-94'
```

## TIME

***Description:*** Time literal value.

***Syntax:***

```
[TIME] '<hour>:<minute>:<second>[.[<fractional_second>]]'
```

***Key words and parameters:***

| | |
|---|---|
| *<hour>* | a two-digit number between 00 and 23. |
| *<minute>* | a two-digit number between 00 and 59. |
| *<second>* | a two-digit number between 00 and 59. |
| *<fractional_second>* | a number containing up to 6 digits. |

***Examples:***

```
'23:00:00'
TIME '23:00:00.'
TIME '23:01:59.134343'
```

## TIMESTAMP

***Description:*** Timestamp literal value.

***Syntax:***

```
TIMESTAMP '<DATE_literal_value> <TIME_literal_value>'
```

***Key words and parameters:***

*<DATE_literal_value>* a Date literal.

*<TIME_literal_value>* a Time literal.

***Comments:*** In a timestamp literal, there is exactly one space character between the Date literal and the Time literal.

***Examples:***

```
TIMESTAMP '1994-11-07 23:00:00'
'94-06-01 12:02:00'
```

***Examples:***

```
CHAR (10)
```

## Pseudocolumns

## LEVEL

***Description:*** The LEVEL pseudocolumn can be used in SELECT statements that perform hierarchical queries. For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root node, 2 for a child of a root, and so on. A root node is the highest node within an inverted tree. A child node is any non-root node, a parent node is any node that has children, and a leaf node is any node without children.

In Personal Oracle Lite, LEVEL can only be used in the select list or WHERE clause of a SELECT statement. It cannot appear in the CONNECT BY clause. The maximum number of levels that can be returned by a hierarchical query is 32. For more information on using the LEVEL pseudocolumn, see the SELECT command in Chapter 2.

***Examples:*** The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is `PRESIDENT`. The child rows of a parent row are defined to be those who have the employee number of the parent row as their manager number.

```
SELECT LPAD(' ',2*(LEVEL-1)) || ename org_chart,
                empno, mgr, job
        FROM emp
        START WITH job = 'PRESIDENT'
        CONNECT BY PRIOR empno = mgr
ORG_CHART         EMPNO        MGR JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
JONES              7566       7839 MANAGER
```

```
SCOTT           7788       7566 ANALYST
ADAMS           7876       7788 CLERK
FORD            7902       7566 ANALYST
SMITH           7369       7902 CLERK
BLAKE           7698       7839 MANAGER
ALLEN           7499       7698 SALESMAN
WARD            7521       7698 SALESMAN
MARTIN          7654       7698 SALESMAN
TURNER          7844       7698 SALESMAN
JAMES           7900       7698 CLERK
CLARK           7782       7839 MANAGER
MILLER          7934       7782 CLERK
```

## C Data Types.

This section briefly describes the C data types to use when accessing the Personal Oracle Lite database. Refer to your C programming documentation for additional information on these data types. Type names of the form SQL_C_* are ODBC identifiers for the type of data in the client-side program.

### CHAR

***Description:*** Fixed length character string type.

***Syntax:*** `char <variable_name> (<size>);`

***Key words and parameters:***

`<variable_name>` name by which the program knows this string.

`<size>` length in bytes to allocate.

***ODBC type:*** SQL_C_CHAR normally. May also be used for SQL_C_TINYINT, SQL_C_UTINYINT, or SQL_C_STINYINT.

***Comments:*** char strings are typically terminated by a null (0) byte, but not always.

When using a null terminated char string to store a value in the database, you may specify that the length of the string is SQL_NTS. Otherwise, it is important to specify the actual length of the string, for example, in the cbValueMax and pcbValue arguments in the SQL Bind Parameter ( ) call.

***Examples:*** `char name (20);`

## SHORT

***Description:*** Short integer or 16-bit integer.

***Syntax:*** `short <variable_name>;`

***Key words and parameters:D*** *<variable_name>* variable typically stored in 2 bytes of memory.

***Examples:*** `short g;`

## LONG

***Description:*** A long integer or a 32-bit integer.

***Syntax:*** `long <variable_name>`

***Key words and parameters:*** *<variable_name>* variable typically stored in 4 bytes of memory.

***ODBC type:*** SQL_C_NTS if null terminated

***Comments:*** Char strings are typically terminated by a null (0) byte, but not always. When using a null terminated char string to store a value in the database, you may specify that the variable type is SQL_C_NTS. Otherwise, you must specify the actual length of the string, for example in the cbValueMax or pcbValue arguments in the SQL Bind Parameter ( ) call.

***Examples:*** `long j;`

## FLOAT

***Description:*** Floating point number type.

***Syntax:*** `float <variable_name>;`

***ODBC type:*** SQL_C_FLOAT

***Key words and parameters:*** *<variable_name>* variable typically stored in 4 bytes of memory.

***Examples:*** `float k;`

## DOUBLE

***Description:*** Double precision floating point number type.

***Syntax:*** `double <variable_name>;`

***ODBC type:*** SQL_C_DOUBLE

***Key words and parameters:*** *<variable_name>* variable typically stored in 8 bytes of memory.

***Examples:***

```
double wavelength;
```

CHAPTER

# 4

# SQL Functions, Predicates, and Expressions

This chapter describes the following:

- SQL functions
- predicates
- expressions

## SQL Functions

A function in ODBC or SQL-92 is any expression in a SQL statement that returns a value. Each function, predicate, and expression described in this chapter can be defined by either the SQL-92, ODBC, or Oracle standard. Many functions are defined by more than one standard, and sometimes with a different name or slightly different functionality. All are legal SQL statements with Personal Oracle Lite.

**Note:** As a general rule, when any input argument of a function is the NULL value, the resulting value of the function is the NULL value. Similarly, when any input argument of a predicate is the NULL value, the resulting value of the predicate is UNKNOWN.

### CASE Function

***Description:*** Specifies a conditional value.

***Syntax:***

```
{fn IFNULL(<value_expression1>, <value_expression2>) }      [ODBC]

CASE <case_operand> <simple_when_clause_list> [ELSE <case_result>] END
[SQL-92]

CASE <searched_when_clause_list> [ELSE <case_result>] END   [SQL-92]

DECODE(<case_operand>, <decode_when_then_list> [, <case_result>] )
       [Oracle]

-------------------------------------------------------------------------

<simple_when_clause_list> ::=

    WHEN <value_expression> THEN <case_result>

   | <simple_when_clause_list>

    WHEN <value_expression> THEN <case_result>

<search_when_clause_list> ::=

    WHEN <search_condition> THEN <case_result>

   | <search_when_clause_list>

    WHEN <search_condition> THEN <case_result>

<decode_when_then_list> ::=

    <value_expression>, <case_result>

   | <decode_when_then_list>, <value_expression>, <case_result>

<case_result> ::= <value_expression> | NULL
```

***Key words and parameters:***

*<value_expression1>*

the return value if its value is not NULL.

*<value_expression2>*

the return value if <value_expression1> is NULL.

*<case_operand>*

a case value (*<value expression>*) to be matched with a given *<value_expression>*. Its corresponding *<case_result>* is returned if a match occurs.

*<case_result>*

one of the possible return values.

***Comments:*** In a case expression, the value of *<case_result>,* if present, is returned when all other conditions are not satisfied.

***Examples:***

```
{fn IFNULL(Employee.Name, 'Unknown') }

CASE Employee.Dept
       WHEN 1 THEN 'S & M'
       WHEN 2 THEN 'R & D'
       ELSE 'G & A'
END

CASE WHEN Employee.Dept = 1 THEN 'S & M'
       WHEN Employee.Dept = 2 THEN 'R & D'
       ELSE 'G & A'
END

DECODE(Employee.Dept, 1, 'S & M', 2, 'R & D', 'G & A')
```

## CAST (Type Conversion) Function

***Description:*** Converts data from one type to another type.

***Syntax:***

```
CAST ( <source_operand> AS <data_type> )                    [SQL-92]
```

***Key words and parameters:***

*<source_operand>*

a value expression or NULL.

*<data_type>*

the type of target.

***Comments:*** Valid combinations of type conversion are tabulated below (V: valid; R: with restrictions; Blank: invalid):

| | EN | AN | VC | FC | D | T | TS | YM | DT |
|---|---|---|---|---|---|---|---|---|---|
| EN | V | V | V | V | | | | R | R |
| AN | V | V | V | V | | | | | |
| C | V | V | R | R | V | V | V | V | V |
| D | | | V | V | V | | V | | |
| T | | | V | V | | V | V | | |
| TS | | | V | V | V | V | V | | |
| YM | R | | V | V | | | | V | |
| DT | R | | V | V | | | | | V |

EN = Number with precision
AN = Float
C = fixed or variable length character
D = date
T = time
TS = timestamp
YM = year-month interval
DT = day-time interval

| If | Then |
|---|---|
| *<source_operand>* is an exact numeric and *<data_type>* is an interval | the interval contains a single datetime field |
| *<source_operand>* is an interval and *<data_type>* is an exact numeric | the interval contains a single datetime field |
| *<source_operand>* is a character string and *<data_type>* specifies a character string | their character repertoire is the same |
| *<data_type>* is numeric and the result can not be represented without losing leading significant digits | the following exception is raised: data-exception, numeric value out of range |

***Examples:***

```
CAST('0' AS INTEGER)
CAST(0 AS REAL)
CAST(1E0 AS NUMERIC(12, 2))
CAST(CURRENT_TIMESTAMP AS VARCHAR(30))
```

## Datetime Extraction Predicates

***Description:*** Extracts the datetime or time zone field from a datetime or interval.

***Syntax:***

```
{ fn <odbc_extract_field> ( <value_expression> ) }          [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn <odbc_extract_field>
       ( < value_expression > ) *)-                          [ODBC]
EXTRACT ( <sql_extract_field> FROM <value_expression> )     [SQL-92]
```

***Key words and parameters:***

*<odbc_extract_field>*
YEAR, MONTH, DAYOFMONTH, HOUR, MINUTE, or SECOND.

*<sql_extract_field>*
YEAR, MONTH, DAY, HOUR, MINUTE or SECOND .

*<value_expression>*
a datetime or an interval from which value is extracted.

***Comments:*** The result has the same sign as the source value expression.

***Examples:***

```
{fn YEAR( {fn CURDATE()} )}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn YEAR({fn CURDATE()})
*)--
EXTRACT(YEAR FROM CURRENT_DATE)
```

## Day-of-Week Predicates

***Description:*** Returns the day of the week as an integer.

***Syntax:***

```
{ fn DAYOFWEEK ( <value_expression> ) }                     [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn DAYOFWEEK
       ( <value_expression> ) *)--                          [ODBC]
```

***Key words and parameters:***

*<value_expression>*
a date on which the weekday is computed.

***Comments:*** The value is between 1 and 7, where 1 represents Sunday.

***Examples:***

```
{fn DAYOFWEEK({CURDATE()})}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn
DAYOFWEEK({CURDATE()}) *)--
```

## Day-of-Year Predicates

***Description:*** Returns the day of the year as an integer.

***Syntax:***

```
{ fn DAYOFYEAR ( <value_expression> ) }                     [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn DAYOFYEAR
```

```
        ( <value_expression> ) *)--                          [ODBC]
```

***Key words and parameters:***

*<value_expression>*
a date on which the day of the year is computed.

***Comments:*** The result is between 1 and 366.

***Examples:***

```
{fn DAYOFYEAR ({CURDATE()})}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn DAYOFYEAR
       ({CURDATE()}) *)--
```

## Datetime Value Functions

***Description:*** Specifies a datetime value.

***Syntax:***

```
{fn TIMESTAMPDIFF (<odbc_interval>, <value_exp1>, <value_exp2>)}[ODBC]
{ fn TIMESTAMPADD ( <odbc_interval>, <value_exp1>, <value_exp2> ) [ODBC]
<value_expression> + <value_expression>                      [SQL-92]
<value_expression> - <value_expression>                      [SQL-92]
```

***Key words and parameters:***

*<odbc_interval>*
one of the following key words:

| | | |
|---|---|---|
| SQL_TSI_FRAC_SECOND | SQL_TSI_WEEK | SQL_TSI_HOUR |
| SQL_TSI_SECOND | SQL_TSI_MONTH | SQL_TSI_DAY |
| SQL_TSI_MINUTE | SQL_TSI_QUARTER | SQL_TSI_YEAR |

*<value_exp1>*
an integer.
*<value_exp2>*
a timestamp.
*<value_expression>*
an operand.

***Comments:*** When the operator is '+', one of the operands must be a datetime, and the other must be an interval. When the operator is '-', the first operand must be a datetime and the second must be an interval. The value of the result is based on the natural rules for the Gregorian calendar. If the type of an operand is `TIME`, then arithmetic on the `HOUR` field is done modulo 24. If the type of an operand is a year-month interval, then the `DAY` field of the result is the same as that of the other operand.

***Examples:***

```
{fn TIMESTAMPDIFF(SQL_TSI_DAY, 1, {fn NOW()})}
--(* VENDOR(vendor_name), PRODUCT(product_name)
        fn TIMESTAMPADD(SQL_TSI_DAY, 1, {fn NOW()}) *)--
CURRENT_DATE - INTERVAL '1' DAY
CURRENT_DATE + INTERVAL '1' DAY
```

## Interval Value Functions

***Description:*** Specifies a time interval value.

***Syntax:***

```
[SQL-92]
- INTERVAL numeric_expr <interval_qualifier>
+ INTERVAL numeric_expr <interval_qualifier>
datetime_expr + INTERVAL num
<value_expression> - <value_expression>
<value_expression> * <value_expression>
<value_expression> / <value_expression>
( <value_expression> - <value_expression> ) <interval_qualifier>
-----------------------------------------------------------------------
<interval_qualifier> ::=
    YEAR [ <precision> ]
  | MONTH [ <precision> ]
  | DAY [ <precision> ]
  | HOUR [ <precision> ]
  | MINUTE [ <precision> ]
  | SECOND [ <precision_scale> ]
  | YEAR [ <precision> ] TO MONTH
  | DAY [ <precision> ] TO HOUR
  | DAY [ <precision> ] TO MINUTE
  | DAY [ <precision> ] TO SECOND [ <precision> ]
  | HOUR [ <precision> ] TO MINUTE
  | HOUR [ <precision> ] TO SECOND [ <precision> ]
  | MINUTE [ <precision> ] TO SECOND [ <precision>
```

***Key words and parameters:***

*<value_expression>*
an operand.

*<interval_qualifier>*
an interval qualifier.

*<precision>*
a number enclosed in parentheses, specifying leading datetime field precision.

*<precision_scale>*
a pair of numbers enclosed in parentheses and separated by a comma, specifying leading field precision and fractional seconds precision, or a number enclosed in parentheses, specifying leading field precision.

***Comments:*** If the operator is '*', then one operand is an interval and the other is a numeric. If the operator is '/', then the first operand is an interval and the second operand is a numeric. If the last form is specified, then the *<value_expression>* is a datetime for both.

***Examples:***

```
- INTERVAL -'1' DAY

+ INTERVAL '1' DAY

INTERVAL '1' YEAR + INTERVAL '3' MONTH

INTERVAL '1-6' YEAR TO MONTH - INTERVAL '3' MONTH

3 * INTERVAL '3' MONTH

INTERVAL '9' MONTH / 3

(CURRENT_DATE - DATE '1994-01-01') DAY
```

## Length Predicates

***Description:*** Returns the bit, byte, or character length of a string.

***Syntax:***

```
{ fn LENGTH ( <value_expression> ) }                          [ODBC]

--(* VENDOR(vendor_name), PRODUCT(product_name) fn LENGTH (
<value_expression> ) *)--                                     [ODBC]

BIT_LENGTH ( <value_expression> )                             [SQL-92]

CHAR_LENGTH ( <value_expression> )                            [SQL-92]

OCTET_LENGTH ( <value_expression> )                           [SQL-92]
```

***Key words and parameters:***

*<value_expression>*
a string expression.

***Comments:*** LENGTH returns the number of characters in *<value_expression>*.

BIT_LENGTH, CHAR_LENGTH, and OCTET_LENGTH returns the length of *<value_expression>* in bits, characters, or octets, respectively.

***Examples:***

```
{fn LENGTH('string')}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn LENGTH('string') *)--
BIT_LENGTH('string')
CHAR_LENGTH('string')
OCTET_LENGTH('string')
```

## Numeric Value Functions

***Description:*** Specifies a numeric value.

***Syntax:***

```
- <value_expression>
+ <value_expression>
<value_expression> + <value_expression>
<value_expression> - <value_expression>
<value_expression> * <value_expression>
<value_expression> / <value_expression>
```

***Key words and parameters:***

*<value_expression>*
an operand.

***Comments:*** A numeric value expression is evaluated from left to right; the unary '+' or '-' is evaluated before other operators, and '*' or '/' is evaluated before binary '+' or '-'.

If the data type of either operand is an approximate numeric, the data type of the result is an approximate numeric; otherwise, it's an exact numeric, with precision and scale determined as follows:

- Let S1 and S2 be the scales of the first and second operands, respectively. Let P1 and P2 be the precisions of the first and second operands, respectively.
- The scale of the result of addition and subtraction is MAX(S1, S2). The precision of the result of addition and subtraction is MAX(P1-S1, P2-S2) + MAX(S1, S2).
- The scale of the result of multiplication is S1 + S2. The precision of the result of multiplication is P1 + P2.
- The scale of the result of division is S1. The precision of the result of division is P1 + S2.

In all cases, the precision of the result will not exceed 24 for exact numeric types. If the value of a divisor is 0, then an exception is raised: data exception, division by zero.

If the result cannot be represented without losing leading significant digits, then the following exception is raised: data exception, numeric value out of range.

***Examples:***

```
1 / 3          returns 0
-(1+1.00/3.00)        returns - 1.33
180 / (2 * 3.1415927E0)returns 28
```

**Note:** The precision and scale of the result depend on the operands, not on the column into which the result may be inserted, as in Oracle. To achieve Oracle results, you may have to CAST the first operand, for example,

```
CAST (1 AS NUMERIC (12,6)) /3 returns .333333
```

## Position Predicates

***Description:*** Returns the starting position of the first occurrence of a substring in a string.

***Syntax:***

```
{ fn LOCATE ( <substring_value_expression> ,
       <value_expression>[, <start_len_cnt> ] ) }          [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn LOCATE (
       <substring_value_expression> , <value_expression>
       [,<start_len_cnt> ] ) *)--                          [ODBC]
POSITION ( <substring_value_expression>
       IN <value_expression> )                             [SQL-92]
```

***Key words and parameters:***

*<value_expression>*
a source string to search in.

*<substring_value_expression>*
a substring to search for.

*<start_len_cnt>*
the starting position for the search.

***Comments:*** If the length of *<substring_value_expression>* is 0, the result is 1. If *<substring_value_expression>* occurs in *<value_expression>,* the result is the position of the first character of *<substring_value_expression>*; otherwise, the result is 0. If *<start_len_cnt>* is omitted, the function starts the search from position 1.

***Examples:***

```
{fn LOCATE('ring', 'string', 1)}

--(* VENDOR(vendor_name), PRODUCT(product_name) fn
       LOCATE('ring', 'string', 1) *)--

POSITION('ring' IN 'string')
```

## Quarter Predicates

***Description:*** Returns the quarter of a date as an integer.

***Syntax:***

```
{ fn QUARTER ( <value_expression> ) }                          [ODBC]

--(* VENDOR(vendor_name), PRODUCT(product_name) fn QUARTER
       ( <value_expression> ) *)--                             [ODBC]
```

***Key words and parameters:***

*<value_expression>*
a date on which the quarter is computed.

***Comments:*** The result is between 1 and 4; where 1 represents the first quarter of the year, Jan. 1 through Mar. 31.

***Examples:***

```
{fn QUARTER({CURDATE()})}

--(* VENDOR(vendor_name), PRODUCT(product_name) fn QUARTER({CURDATE()})
*)--
```

## Set Predicates

***Description:*** Returns a value derived by applying a set predicate to an argument. They are also called aggregate predicates.

***Syntax:***

```
COUNT(*)

<set_function> ([<set_quantifier>] <value_expression>)
```

***Key words and parameters:***

*<value_expression>*
an argument to which a set function is applied. It should not contain another set function or a subquery.

***Comments:*** If *<set_quantifier>* is not specified, ALL is implied.

The data type of *<value_expression>* will not be character string or datetime.

If `COUNT(*)` is specified, the result is the cardinality of the table or a group in a grouped table. Otherwise, the set function is applied to *<value_expression>*. Null values are eliminated from a single-column table. If `DISTINCT` is specified, duplicate values are also eliminated from a single-column table.

***Examples:***

```
COUNT (*)
COUNT (DISTINCT Name)
SUM (ALL Sales)
MIN (Age)
MAX (Salary)
AVG (Price)
```

## Substring Predicates

***Description:*** Constructs and returns a substring from a string.

***Syntax:***

```
{fn SUBSTRING ( <value_expression> ,<start>, <len_cnt> ))   [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn SUBSTRING (
       <value_expression> , <start> , <len_cnt> ) *)--      [ODBC]
SUBSTRING ( <value_expression>
       FROM <start> [, FOR <len_cnt> ] )                    [SQL-92]
```

***Key words and parameters:***

*<value_expression>*

a string expression.

*<start>*

the starting position in a source string.

*<len_cnt>*

the length of the substring to be constructed.

***Examples:***

```
{fn SUBSTRING('string', 2, 1)}
       --(* VENDOR(vendor_name), PRODUCT(product_name) fn
       SUBSTRING('string', 2, 1) *)--
       SUBSTRING('string', FROM 2 FOR 1)
```

## Trim Predicates

***Description:*** Removes leading and/or trailing blanks (or other characters) from a string.

***Syntax:***

```
{ fn LTRIM ( <value_expression> ) }                         [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn LTRIM
       ( <value_expression> ) *)--                          [ODBC]
{ fn RTRIM ( <value_expression> ) }                         [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn RTRIM
       ( <value_expression> ) *)--                          [ODBC]
```

```
TRIM ( [[<trim_spec>] [<trim_char>]
       FROM ] <value_expression> )                          [SQL-92]
```

***Key words and parameters:***

*<value_expression>*
a string expression.

*<trim_spec>*
a specification: LEADING, TRAILING, or BOTH.

*<trim_char>*
the character to be removed.

***Comments:*** LTRIM returns a substring of *<value_expression>* with leading blanks removed. RTRIM returns the characters of *<value_expression>* with trailing blanks removed. If *<trim_spec>* is omitted, then BOTH is implied. If *<trim_char>* is omitted, then '' is implied.

***Examples:***

```
{fn LTRIM('  string')}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn LTRIM('  string') *)--
{fn RTRIM('string  ')}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn RTRIM('string  ') *)--
TRIM(BOTH 's' FROM 'strings')
TRIM('s' FROM 'strings')
TRIM(TRAILING FROM 'strings  ')
```

## Week Predicates

***Description:*** Returns the week of the year as an integer.

***Syntax:***

```
{ fn WEEK ( <value_expression> ) }                          [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn WEEK
( <value_expression> ) *)--                                 [ODBC]
```

***Key words and parameters:***

*<value_expression>*
a date on which the week is computed.

***Comments:*** The result is between 1 and 53.

***Examples:***

```
{fn WEEK({CURDATE()})}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn WEEK({CURDATE()}) *)-
-
```

### String Value Functions

***Description:*** Concatenates two strings to form a new string.

***Syntax:***

```
CONCAT(stringa, stringb)                                          [Oracle]
{ fn CONCAT ( <value_expression>,<value_expression> ) }           [ODBC]
<stringa> || <stringb>                                            [SQL-92]
```

***Key words and parameters:***

*<stringa>* and *<stringb>*
a string to be concatenated.

***Examples:***

```
{fn CONCAT('a ', 'string')}

--(* VENDOR(vendor_name), PRODUCT(product_name) fn CONCAT('a ',
'string') *)--

'a ' || 'string'
```

### Value Functions

***Description:*** Specifies a value.

***Syntax:***

```
<numeric_value_expression>

<string_value_expression>

<datetime_value_expression>

<interval_value_expression>
```

***Key words and parameters:***

*<numeric_value_expression>*
an expression specifying a numeric value. See Numeric Value Function.

*<string_value_expression>*
an expression specifying a string value. See String Value Function.

*<datetime_value_expression>*
an expression specifying a datetime value. See Datetime Value Function.

*<interval_value_expression>*
an expression specifying an interval value. See Interval Value Function.

***Examples:*** See subsections on Numeric Value Function, String Value Function, Datetime Value Function, and Interval Value Function.

### CEIL (Number Function)

***Syntax:***

```
CEIL (n)                                                          [Oracle]
```

***Purpose:*** Returns smallest integer greater than or equal to *n*.

***Example:***

```
SELECT CEIL(15.7) "Ceiling"
  FROM DUAL

   Ceiling
----------
       16
```

## FLOOR (Number Function)

***Syntax:*** `FLOOR (n)` [Oracle]

***Purpose:*** Returns largest integer equal to or less than *n*.

***Example:***

```
SELECT FLOOR(15.7) "Floor"
  FROM DUAL

   Floor
--------
     15
```

## MOD (Number Function)

***Syntax:*** `MOD (m,n)` [Oracle]

***Purpose:*** Returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

***Example:***

```
SELECT MOD(11,4) "Modulus"
  FROM DUAL

   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the MOD function with this formula: m - n * FLOOR(m/n)

The following statement illustrates the difference between the MOD function and the classical modulus:

```
SELECT m, n, MOD(m, n),
m - n * FLOOR(m/n) "Classical Modulus"
  FROM test_mod_table

  M   N MOD (M,N) Classical Modulus
--- ---- -------- --------- -------
 11   4        3
-11   4       -3                 1
 11  -4        3                -1
```

```
-11  -4       -3                -3
```

## ROUND (Number Function)

***Syntax:*** `ROUND (n[,m])` `[Oracle]`

***Purpose:*** Returns *n* rounded to *m* places right of the decimal point; if *m* is omitted, to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

***Example:***

```
SELECT ROUND(15,193,1) "Round"
  FROM DUAL

   Round
--------
    15,2
```

***Example:***

```
SELECT ROUND(15,193,-1) "Round"
  FROM DUAL

   Round
--------
      20
```

## ASCII (Character Function)

***Syntax:*** `ASCII(char)`

***Purpose:*** Returns the decimal representation in the database character set of the first byte of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value.

***Example:***

```
SELECT ASCII('Q')
    FROM DUAL

ASCII('Q')
----------
        81
```

## CHR (Character Function)

***Syntax:*** `CHR(n)`

***Purpose:*** Returns the character having the binary equivalent to *n* in the database character set.

***Example:*** This example uses nesting to concatenate three character strings:

```
SELECT CHR(67)||CHR(68)||CHR(84) "Dog"
  FROM DUAL
Dog
---
CAT
```

## CONCAT (Character Function)

***Syntax:*** `CONCAT(char1, char2`

***Purpose:*** Returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||). For additional information on this operator, refer to *Oracle7 Server SQL Reference*.

***Example:*** This example uses nesting to concatenate three character strings:

```
SELECT CONCAT( CONCAT(ename, ' is a '), job) "Job"
  FROM emp
  WHERE empno = 7900

Job
-------------------------
JAMES is a CLERK
```

## INITCAP (Character Function)

***Syntax:*** `INITCAP(char)`

***Purpose:*** Returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

***Example:***

```
SELECT INITCAP('the soap') "Capitals"
  FROM DUAL

Capitals
--------
The Soap
```

## INSTR (Character Function)

***Syntax:*** `INSTR(char1,char2[,n[,m]])`

***Purpose:*** Searches the string argument *char1*, beginning with its *n*th character, for the *m*th occurrence of string argument *char2*, where *n* and *m* are numeric arguments. Returns the position in *char1* of the first character of this occurrence.

***Example:***

```
SELECT INSTR('CORPORATE FLOOR','OR',3,2) "Instring"
  FROM DUAL

Instring
--------
      14
```

## INSTRB (Character Function)

***Syntax:*** `INSTRB(char1,char2[,n[,m]])`

***Purpose:*** The same as INSTR, except that *n* and the return value are expressed in bytes, rather than in characters. For a single-byte database character set, INSTRB is equivalent to INSTR.

***Example:***

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes"
  FROM DUAL

Instring in bytes
-----------------
               27
```

## LENGTH (Character Function)

***Syntax:***

```
LENGTH(char)

{fn LENGTH (char)}

BIT_LENGTH (char)

CHAR_LENGTH (char)

OCTET_LENGTH (char)
```

***Purpose:*** Returns the length of *char* in characters. If *char* has datatype CHAR, the length includes all trailing blanks. If *char* is null, the function returns null. `BIT_LENGTH`, `CHAR_LENGTH`, and `OCTET_LENGTH` return the length of *char* in bits, characters, or octets, respectively.

***Example:***

```
SELECT LENGTH('CANDIDE') "Length in characters"
  FROM DUAL

Length in characters
--------------------
                   7
```

## LENGTHB (Character Function)

***Syntax:***

```
LENGTHB(char)
```

***Purpose:*** Returns the length of *char* in bytes. If char is null, this function returns null. For a single-byte database character set, LENGTHB is equivalent to LENGTH.

***Example:*** Assume a double-byte database character set:

```
SELECT LENGTHB('CANDIDE') "Length in bytes"
  FROM DUAL

Length in bytes
---------------
             14
```

## LPAD (Character Function)

***Syntax:*** `LPAD(char1,n [,char2])`

***Purpose:*** Returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multi-byte character sets, the display length of a character string can differ from the number of characters in the string.

***Example:***

```
SELECT LPAD('Page1',15,'*.') "LPAD example"
        FROM DUAL

LPAD example
---------------
*.*.*.*.*Page 1
```

## LTRIM (Character Function)

***Syntax:*** `LTRIM(char1 [,set])`

***Purpose:*** Removes characters from the left of *char*, with all the leftmost characters that appear in *set* removed; *set* defaults to a single blank.

***Example:***

```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
        FROM DUAL

LTRIM example
-------------
XxyLAST WORD
```

## REPLACE (Character Function)

***Syntax:*** `REPLACE(char, search_string[,replacement_string])`

***Purpose:*** Returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, all occurrences of *search_string* are removed. If *search_string* is null, *char* is returned. This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single character, one to one, substitution. REPLACE allows you to substitute and remove entire character strings.

***Example:***

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
  FROM DUAL

Changes
```

```
--------------
BLACK and BLUE
```

## RPAD (Character Function)

***Syntax:*** `LPAD(char1,n [,char2])`

***Purpose:*** Returns *char1*, right-padded to length *n* with *char2* replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multi-byte character sets, the display length of a character string can differ from the number of characters in the string.

***Example:***

```
SELECT RPAD('ename',12,'ab') "RPAD example"
    FROM emp
    WHERE ename = 'TURNER'

RPAD example
------------
TURNER     ab
```

## RTRIM (Character Function)

***Syntax:*** `RTRIM(char [,set]`

***Purpose:*** Returns *char*, with all the rightmost characters that appear in *set* removed; set defaults to a single blank. RTRIM works similarly to LTRIM.

***Example:***

```
SELECT RTRIM('TURNERyxXxy','xy') "RTRIM e.g."
  FROM DUAL

RTRIM e.g
---------
TURNERyxX
```

**ODBC Function:**

`{fn RTRIM (char)}` (trims leading blanks)

## SUBSTR (Character Function)

***Syntax:*** `SUBSTR(char, m [,n])`

***Purpose:*** Returns a portion of *char*, beginning at character *m*, *n* characters long. If *m* is 0, it is treated as 1. If *m* is positive, Personal Oracle Lite counts from the beginning of *char* to find the first character. If *m* is negative, Personal Oracle Lite counts backwards from

the end of *char*. If *n* is omitted, Personal Oracle Lite returns all characters to the end of *char*. If *n* is less than 1, a null is returned.

***Example:***

```
SELECT SUBSTR('ABCDEFG',3,4) "Subs"
  FROM DUAL

Subs
----
CDEF

SELECT SUBSTR('ABCDEFG',-5,4) "Subs"
  FROM DUAL

Subs
----
CDEF
```

## SUBSTRB (Character Function)

***Syntax:***

```
SUBSTRB(char, m [,n])
```

***Purpose:*** The same as SUBSTR, except that the arguments m and n are expressed in bytes, rather than in characters. For a single-byte database character set, SUBSTRB is equivalent to SUBSTR.

Assume a double-byte database character set:

***Example:***

```
SELECT SUBSTRB('ABCDEFG',5,4) "Substring with bytes"
  FROM DUAL

Sub
---
EFG
```

## TRANSLATE (Character Function)

***Syntax:***

```
TRANSLATE(char, from, to)
```

***Purpose:*** Returns *char* with all occurrences of each character in *from* replaced by its corresponding character in *to*. Characters in *char* that are not in *from* are not replaced. The argument *from* can contain more characters than *to*. In this case, the extra characters at the end of *from* have no corresponding characters in *to*. If these extra characters appear in *char*, they are removed from the return value. You cannot use an empty string for *to* to remove all characters in *from* from the return value. Personal Oracle Lite interprets the empty string as null, and when this function has a null argument, it returns null.

The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012...9' are translated to '9':

***Example:***

```
SELECT TRANSLATE('2KRW229', '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXX') "Licence"
  FROM DUAL

Translate example
-----------------
9XXX999
```

The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229', '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'0123456789')
"Translate example"
  FROM DUAL

Translate example
-----------------
2229
```

## ADD_MONTHS (Date Function)

***Syntax:*** `ADD_MONTHS(d,n)`

***Purpose:*** Returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

***Example:***

```
SELECT TO_CHAR(
    ADD_MONTHS(hiredate,1),
    'DD-MM-YYYY' "Next month"
    FROM emp
    WHERE ename = 'SMITH'

Next Month
-----------
17-JAN-1981
```

## LAST_DAY (Date Function)

***Syntax:*** `LAST_DAY(d)`

***Purpose:*** Returns the date of the last day of the month that contains *d*. You might use this function to determine how many days are left in the current month.

***Example:***

```
SELECT SYSDATE,
    LAST_DAY(SYSDATE) "Last",
    LAST_DAY(SYSDATE) - SYSDATE "Days Left"
    FROM DUAL
```

```
SYSDATE   Last       Days Left
--------- --------- ----------
10-APR-95 30-APR-95         20
```

```
SELECT TO_CHAR(
    ADD_MONTHS(
        LAST_DAY(hiredate),5),
        'DD-MON-YYYY') "Five months"
    FROM emp
    WHERE ename = 'MARTIN'


Five months
-----------
28-FEB-1982
```

## MONTHS_BETWEEN (Date Function)

***Syntax:*** `MONTHS_BETWEEN(d1, d2)`

***Purpose:*** Returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer; otherwise Personal Oracle Lite calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

***Example:***

```
SELECT MONTHS_BETWEEN(

    TO_DATE('02-02-1995','MM-DD-YYYY'),
    TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
    FROM DUAL

    Months
----------
1.03225806
```

## NEXT_DAY (Date Function)

***Syntax:*** `NEXT_DAY(d, char)`

***Purpose:*** Returns the date of the first weekday named by *char* that is later than the date *d*. The argument *char* must be a day of the week in your session's date language. The return value has the same hours, minutes, and seconds component as the argument *d*.

***Example:***

```
SELECT NEXT_DAY('15-MAR-92','TUESDAY') "NEXT DAY"
    FROM DUAL

NEXT DAY
---------
17-MAR-92
```

## ROUND (Date Function)

***Syntax:*** `ROUND(d[,fmt])`

***Purpose:*** Returns *d* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is rounded to the nearest day.

***Example:***

```
SELECT ROUND(TO_DATE('27-OCT-92'),'YEAR')
"FIRST OF THE YEAR"
    FROM DUAL

FIRST OF THE YEAR
-----------------
01-JAN-93
```

The following table lists the format models to be used with the ROUND (and TRUNC) date function, and the units to which it rounds dates. The default model, 'DD', returns the date rounded to the day with a time of midnight. .

| Format Model | Rounding Unit |
|---|---|
| CC, SCC | Century |
| SYYYY, YYYY<br>YEAR, SYEAR<br>YYY, YY, Y | Year (rounds up on July 1) |
| IYYY, IY<br>IY, I | ISO Year |
| Q | Quarter (rounds up on the sixteenth day of the second month of the quarter) |
| MONTH, MON<br>MM, RM | Month (rounds up on the sixteenth day) |
| WW | Same day of the week as the first day of the year. |
| IW | Same day of the week as the first day of the ISO year. |
| W | Same day of the week as the first day of the month. |
| DDD, DD, J | Day |
| DAY, DY, D | Starting day of the week |
| HH, HH12, HH24 | Hour |
| MI | Minute |

**Note**: The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter NLS_TERRITORY

## SYSDATE (Date Function)

***Syntax:*** `SYSDATE`

***Purpose:*** Returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint. SYSDATE only inserts the date into a Personal Oracle Lite DATE type column, only the time into a TIME column, and both date and time into a TIMESTAMP column.

| SQL-92: | ODBC: |
|---|---|
| CURRENT_DATE | {fn CURDATE()} |
| CURRENT_TIME | {fn CURTIME()} |
| CURRENT_TIMESTAMP | {fn NOW()} |

***Example:***

```
SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS') NOW
  FROM DUAL

NOW
-------------------
10-29-1993 20:27:11.
```

## TRUNC (Date Function)

***Syntax:*** `TRUNC(d[, fmt])`

***Purpose:*** Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is truncated to the nearest day.

***Example:***

```
SELECT TRUNC(TO_DATE('27-OCT-92', 'DD-MON-YY'), 'YEAR') "First Of The
Year"
  FROM DUAL

FIRST OF THE YEAR
-----------------
01-JAN-92
```

The following table lists the format models to be used with the TRUNC (and ROUND) date function, and the units to which it truncatess dates. The default model, 'DD', returns the date truncated to the day with a time of midnight. .

| Format Model | Truncating Unit |
|---|---|
| CC, SCC | Century |
| SYYYY, YYYY<br>YEAR, SYEAR<br>YYY, YY, Y | Year (rounds up on July 1) |

| Format Model | Truncating Unit |
|---|---|
| IYYY, IY<br>IY, I | ISO Year |
| Q | Quarter (rounds up on the sixteenth day of the second month of the quarter) |
| MONTH, MON<br>MM, RM | Month (rounds up on the sixteenth day) |
| WW | Same day of the week as the first day of the year. |
| IW | Same day of the week as the first day of the ISO year. |
| W | Same day of the week as the first day of the month. |
| DDD, DD, J | Day |
| DAY, DY, D | Starting day of the week |
| HH, HH12, HH24 | Hour |
| MI | Minute |

**Note**: The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter NLS_TERRITORY

## TO_CHAR (Conversion Function)

***Purpose:*** Converts a date or number to a value of VARCHAR2 datatype, using the optional format *fmt*.

***Syntax:*** `TO_CHAR(d [, fmt])`

***Key words and parameters:***

<*d*>

Date column or SYSDATE

<*fmt*>

format string

<*n*>

number column or literal

If you omit *fmt*, the argument *d* or *n* is converted to a VARCHAR2 value. For dates, the argument *d* is returned in the default date format. For numbers, the argument *n* is converted to a value exactly long enough to hold its significant digits.

Data literals must be preceded by the DATE keyword when used as arguments to TO_CHAR.

***Example:***

```
TO_CHAR (SYSDATE, 'Day, Month, DD, YYYY')
```

returns

```
"Monday, July 1, 1996"
```

```
TO_CHAR (DATE '1996-07-01', 'Day')
```

returns

```
"Monday"
```

## TO_DATE (Conversion Function)

***Syntax:*** `TO_DATE(`*`char`* `[,` *`fmt`*`])`

***Purpose:*** Converts the character string argument *char* to a value of DATE datatype. The *fmt* argument is a date format specifying the format of *char*.

Do not use the TO_DATE function with a DATE value for the *char* argument. The returned DATE value can have a different century value than the original *char*, depending on *fmt* or the default date format.

Dates in either Oracle format (such as '06-JUN-85' and '6-JUN-1985') or SQL-92 format (such as '1989-02-28') are converted automatically when inserted into a date column.

The following table lists the date format elements.

| Element | Meaning |
|---|---|
| – / , . ; : "text" | Punctuation and quoted text is reproduced in the result. |
| AD, A.D. | AD indicator with or without periods. |
| AM, A.M | Meridian indicator with or without periods. |
| BC, B.C. | BC indicator with or without periods. |
| CC, SCC | Century; "S" prefixes BC dates with "–". |
| D | Day of week (1–7). |
| DAY | Name of day, padded with blanks to length of 9 characters. |
| DD | Day of month (1–31). |

| Element | Meaning |
|---|---|
| DDD | Day of year (1–366). |
| DY | Abbreviated name of day. |
| IW | Week of year (1–52 or 1–53) based on the ISO standard. |
| IYY, IY, I | Last 3, 2, or 1 digit(s) of ISO year. |
| IYYY | 4–digit year based on the ISO standard. |
| HH, HH12 | Hour of day (1–12). |
| HH24 | Hour of day (0–23). |
| MI | Minute (0–59). |
| MM | Month (01–12; JAN = 01) |
| MONTH | Name of month, padded with blanks to length of 9 characters. |
| MON | Abbreviated name of month. |
| Q | Quarter of year (1, 2, 3, 4; JAN–MAR = 1) |
| WW | Week of year (1–53) where week 1 starts on the first day of the year and continues to the seventh day of the year. |
| W | Week of month (1–5) where week 1 starts on the first day of the month and ends on the seventh. |
| PM, P.M. | Meridian indicator with and without periods. |
| RR | Last 2 digits of year; for years in other countries. |
| SS | Second (0–59). |
| SSSSS | Seconds past midnight (0–86399). |
| Y, YYY | Year with comma in this position. |
| YYYY, SYYYY SYEAR, YEAR | 4–digit year; “S” prefixes BC dates with “–”. |
| YYY, YY, Y | Last 3, 2, or 1 digit(s) of year. |

## TO_NUMBER (Conversion Function)

***Syntax:*** `TO_NUMBER(char [,fmt])`

***Purpose:*** Converts the string argument *char* that contains a number in the format specified by the optional format model *fmt* to a return value of the NUMBER datatype.

***Example:***

```
UPDATE emp
 SET sal = sal +
   TO_NUMBER('100.00', '9G999D99')
 WHERE ename = 'BLAKE'
```

The following table lists the number format element values determined by initialization parameters. A number format model can contain only a single decimal character (D) or period (.), but it can contain multiple group separators (G) or commas (,). A number format model must not begin with a comma (,). A group separator or comma cannot appear to the right of a decimal character or period in a number format model.

| Element | Example | Description |
|---|---|---|
| 9 | 9999 | Return value with the specified number of digits with a leading space if positive. Return value with the specified number of digits with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed point number. |
| 0 | 0999<br>9990 | Return leading zeros. Return trailing zeros. |
| $ | $9999 | Return value with a leading dollar sign. |
| B | B9999 | Return blanks for the integer part of a fixed point number when the integer part is zero (regardless of "0"s in the format model). |
| MI | 9999MI | Return negative value with a trailing minus sign "–". Returns positive value with a trailing blank. |
| S | S9999<br>9999S | Return negative value with a leading minus sign "–". Return positive value with a leading plus sign "+". Return negative value with a trailing minus sign "–". Return positive value with a trailing plus sign "+". |
| PR | 9999PR | Return negative value in <angle brackets>. Return positive value with a leading and trailing blank. |
| D | 99D99 | Return a decimal point (that is, a period ".") in the specified position. |
| G | 9G999 | Return a group separator in the position specified. |
| C | C999 | Return the ISO currency symbol in the specified position. |
| L | L999 | Return the local currency symbol in the specified position. |
| , (comma) | 9,999 | Return a comma in the specified position. |

| Element | Example | Description |
|---|---|---|
| . (period) | 99.99 | Return a decimal point (that is, a period ".") in the specified position. |
| EEEE | 9.9EEEE | Return a value using scientific notation. |

## STDDEV

***Syntax:*** `STDDEV([DISTINCT|ALL] x)`

***Purpose:*** Returns standard deviation of *x*, a number. Personal Oracle Lite calculates the standard deviation as the square root of the variance defined for the VARIANCE group function.

***Example:***

```
SELECT STDDEV(sal) "Deviation"
       FROM emp

Deviation
----------
1182.50322
```

## VARIANCE (Group Function)

***Syntax:*** `VARIANCE([DISTINCT|ALL]x)`

***Purpose:*** Returns variance of *x*, a number. Personal Oracle Lite calculates the variance of *x* using this formula:

$$\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2}{n-1}$$

where:

$x_i$ 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.

***Example:***

```
SELECT VARIANCE(sal) "Variance"
       FROM emp

Variance
----------
1389313.87
```

## GREATEST

***Syntax:***

```
GREATEST(expr [,expr] ...)
```

***Purpose:*** Returns the greatest of the list of *exprs*. All *exprs* after the first are implicitly converted to the datatype of the first *exprs* before the comparison. Personal Oracle Lite compares the *exprs* using non-padded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher value. If the value returned by this function is character data, its datatype is always VARCHAR2.

***Example:***

```
SELECT GREATEST('HARRY','HARRIOT','HAROLD') "GREATEST"
       FROM DUAL

GREATEST
--------
HARRY
```

## LEAST

***Syntax:***

```
LEAST(expr [,expr] ...)
```

***Purpose:*** Returns the least of the list of *exprs*. All *exprs* after the first are implicitly converted to the datatype of the first *expr* before the comparison. Personal Oracle Lite compares the *exprs* using non-padded comparison semantics. If the value returned by this function is character data, its datatype is always VARCHAR2.

***Example:***

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST"
       FROM DUAL

LEAST
------
HAROLD
```

## NVL

***Syntax:***

```
NVL(expr1, expr2)

{fn IFNULL (expr1, expr2)}
```

***Purpose:*** If *expr1* is null, returns *expr2*; if *expr1* is not null, returns *expr1*. The arguments *expr1* and *expr2* must be of the same datatype.

***Example:***

```
SELECT ename, NVL(TO_CHAR(COMM),'NOT APPLICABLE') "COMMISSION"
  FROM emp
```

```
  WHERE deptno = 30

ENAME     COMMISSION
--------- -----------
ALLEN     300
WARD      500
MARTIN    1400
BLAKE     NOT APPLICABLE
TURNER    0
JAMES     NOT APPLICABLE
```

## Predicates

A predicate is any variable that returns a Boolean value of TRUE, FALSE, or UNKNOWN. This section descibes the following predicates:

- BETWEEN
- Case Conversion (includes LOWER, UPPER, LCASE, and UCASE)
- Search Conditions
- Comparison
- IN
- LIKE
- NULL
- Quantified Comparison

### BETWEEN Predicates

***Description:*** Tests whether a value is within a range.

***Syntax:***

```
<value_expression> BETWEEN <value_expression> AND <value_expression>
<value_expression> NOT BETWEEN <value_expression> AND <value_expression>
```

***Key words and parameters:***

*<value_expression>*
an operand.

***Comments:*** 'V BETWEEN V1 AND V2 is equivalent to 'V1 <= V AND V <= V2.
'V NOT BETWEEN V1 AND V2 is equivalent to NOT (V BETWEEN V1 AND V2).

***Examples:***

```
interest_rate BETWEEN 0.5 AND 0.8
```

```
SELECT empno from demoemp where empno is between 7566 and 7689

EMPNO
```

```
------
7654
```

## Case Conversion Predicates

***Description:*** Converts a string to all uppercase or all lowercase.

***Syntax:***

```
{ fn LCASE ( <value_expression> ) }                          [ODBC]

--(* VENDOR(vendor_name), PRODUCT(product_name) fn LCASE (
<value_expression> ) *)--                                    [ODBC]

{ fn UCASE ( <value_expression> ) }                          [ODBC]

--(* VENDOR(vendor_name), PRODUCT(product_name) fn UCASE

       ( <value_expression> ) *)--                           [ODBC]

LOWER ( <value_expression> )                                 [SQL-92]

UPPER ( <value_expression> )                                 [SQL-92]
```

***Key words and parameters:***

*<value_expression>*
a source string.

***Comments:*** LCASE and LOWER convert their arguments to all lowercase. UCASE and UPPER convert their arguments to all uppercase.

***Examples:***

```
{fn LCASE('UpperCase')}

--(* VENDOR(vendor_name), PRODUCT(product_name) fn UCASE('lowercase')
*)--

SELECT LOWER('LOWER') from dual

------
lower


SELECT UPPER('UPPER') from dual

------
UPPER
```

## Comparison Predicates

***Description:*** Compares two values and returns TRUE, FALSE, or UNKNOWN.

***Syntax:***

```
<row_value_constructor_element>

       <comp_op> <row_value_constructor_element>
```

***Key words and parameters:***

*<row_value_constructor_element>*

a value.

*<comp_op>*

one of the following: <, <=, =, >, >=, or <>.

***Comments:***

| This... | Means... | This... | Means... |
|---|---|---|---|
| = | equal | <= | less than or equal to |
| <> | not equal | > | greater than |

If either value is NULL, the result is UNKNOWN. If the values are numeric, the comparison is based on their algebraic value. If the values are strings, the comparison is based on lexicographic order.

The comparison of datetimes is based on the interval of or taken from/computed from their difference. The comparison of intervals is based on their corresponding values after conversion to integers in some common base unit.

***Examples:***

```
1 = 1.0

0.5 <> 1/2

'star' < 'string'

CURRENT_DATE <= DATE '2100-12-08'

INTERVAL '1' DAY > INTERVAL '23' HOUR

pi >= 3.14159
```

## IN Predicates

***Description:*** Tests whether a value is in a list.

***Syntax:*** *<value_expression>* [ NOT ] IN ( *<query_spec>* )

***Key words and parameters:***

*<value_expression>*

an operand.

*<query_spec>*

a query specification.

***Comments:*** If any row in *<query_spec>* equals *<value_expression>*, the result is TRUE.

*<value_expression>* NOT IN ( *<query_spec>* ) is the same as NOT ( *<value_expression>* IN ( *<query_spec>* ).

***Examples:*** `'Apple' NOT IN ('Apricot', 'Orange')`

### LIKE Predicates

***Description:*** Tests whether a string matches a pattern string.

***Syntax:***

```
<value_expression> [NOT] LIKE <value_expression2>
        [ ESCAPE <value_expression3> ]
```

***Key words and parameters:***

*<value_expression>*
a source string to be matched against a pattern string.

*<value_expression2>*
a pattern string.

*<value_expression3>*
the escape character.

***Comments:*** Wild card characters '%', '_', and escape characters can appear in the pattern string. '%' matches 0 or more characters in *<value_expression>,* while '_' matches a single character in *<value_expression>.* Blank characters in *<value_expression>* must be matched explicitly.

A '_', '%', or an escape character that immediately follows an escape character is matched literally (that is, it has no special meaning).

***Examples:***

```
ename LIKE '%ER';
ename LIKE '_URNER';
```

### NULL Predicates

***Description:*** Tests if a value is null.

***Syntax:***

```
<column_reference> IS [ NOT ] NULL
```

***Key words and parameters:***

*<column_reference>*
a value to be tested.

***Comments:*** '*<column_reference>* IS NULL' is TRUE if its value is null.

***Examples:***

```
EmpID IS NOT NULL
```

### Quantified Comparison Predicates

***Description:*** Specifies a quantified comparison.

***Syntax:***

```
<row_value_constructor_element> <comp_op> ALL <query_spec>
<row_value_constructor_element> <comp_op> ANY <query_spec>
<row_value_constructor_element> <comp_op> SOME <query_spec>
```

***Key words and parameters:***

*<row_value_constructor_element>*
a *<value_expression>* or *<query_spec>.*

*<comp_op>*
one of the following: <, <=, =, >, >= or <>.

*<query_spec>*
a query specification.

***Comments:***

'=', '<>', '<', '<=', '>', '>=' means equal, not equal, less than, less than or equal to, greater than, greater than or equal to, respectively.

The result of *<row_value_constructor_element> <comp_op>* ALL *(<query_spec>)* is TRUE if *<query_spec>* is empty or if the comparison predicate is TRUE for every row in *<query_spec>*; it is FALSE if the comparison predicate is FALSE for at least one row in *<query_spec>.*

The result of *<row_value_constructor_element> <comp_op>* ANY *(<query_spec>)* is TRUE if the comparison predicate is TRUE for at least one row in *<query_spec>*; it is FALSE if the comparison predicate is FALSE for every row in *<query_spec>*.

The result of *<row_value_constructor_element> <comp_op>* SOME *(<query_spec>)* has the same value as *<row_value_constructor_element> <comp_op>* ANY *(<query_spec>).*

If the predicate is neither TRUE nor FALSE, then the result is UNKNOWN.

***Examples:***

```
Emp_ID = SOME (SELECT ID FROM Employee WHERE Salary >= AVG(Salary))
Salary <= ALL (SELECT Salary FROM Employee WHERE Dept = 'R & D')
```

## Search Conditions

***Description:***

Specifies a Boolean expression that has the value TRUE, FALSE, or UNKNOWN.

***Syntax:***

```
( <predicate> )
NOT <predicate>
<predicate> AND <predicate>
<predicate> OR <predicate>
```

***Key words and***

***parameters:*** *<predicate>*

one of the following predicates: comparison predicate, BETWEEN predicate, IN predicate, LIKE predicate, or NULL predicate.

**Note:** "NOT" is used in an expression as follows:

```
'IS NOT NULL', 'NOT LIKE& string3', and 'NOT BETWEEN'
```

***Comments:*** The evaluation order of the operators is NOT, followed by AND, followed by OR.

NOT(TRUE) is FALSE, NOT(FALSE) is TRUE, and NOT(UNKNOWN) is UNKNOWN.

The truth tables for other Boolean operators are listed below:

| AND | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | FALSE | UNKNOWN |
| FALSE | FALSE | FALSE | FALSE |
| UNKNOWN | UNKNOWN | FALSE | UNKNOWN |

| OR | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | UNKNOWN |
| UNKNOWN | TRUE | UNKNOWN | UNKNOWN |

## Expressions

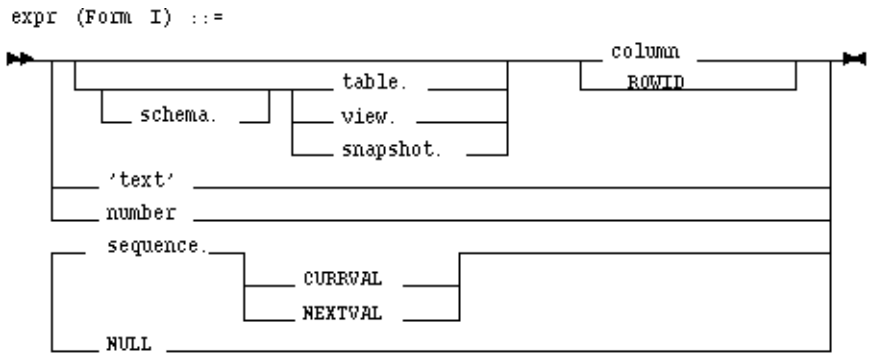
An expression is a combination of one or more values, operators, and SQL functions that evaluates to a value. This section describes the use of expr.

### EXPR

***Purpose:*** To specify an expression of any datatype. You must use this notation whenever *expr* appears in conditions, SQL functions, or SQL commands described in *Oracle7 Server SQL Reference*.

***Syntax:*** Expressions have several forms. Personal Oracle Lite does not accept all forms of expressions in all parts of all SQL commands. The description of each command in Chapter 4 of *Oracle7 Server SQL Reference* documents the restrictions on the expressions in the command.

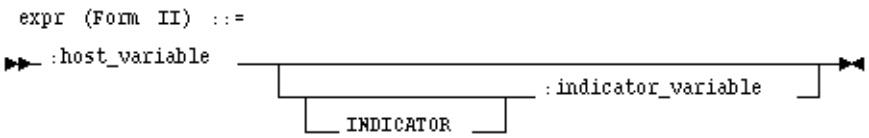
***Form I:*** A column, pseudocolumn, constant, sequence number, or NULL.




**Examples**

```
emp.ename
'this is a text string'
10
```

***Form II:*** A host variable with an optional indicator variable. Note that this form of expression can only appear in embedded SQL statements or SQL statements processed in an Oracle Call Interfaces program.

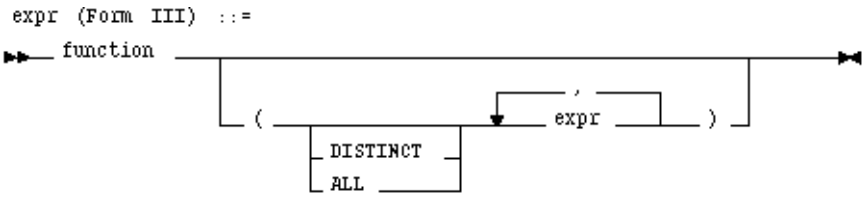



**Examples**

```
:employee_name INDICATOR :employee_name_indicator_var

:department_location
```

***Form III:*** A call to a SQL function.



For information on SQL functions, see the section “SQL Functions” in *Oracle7 Server SQL Reference*.

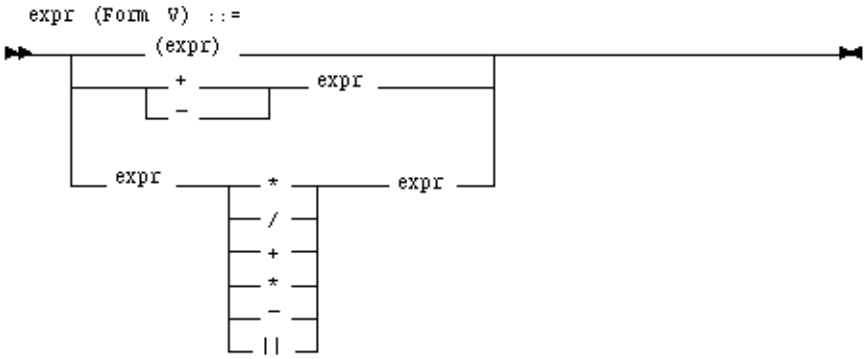
**Examples**

```
LENGTH(‘BLAKE’)

SYSDATE

MOD (23,3)
```

***Form V:*** A combination of other expressions.



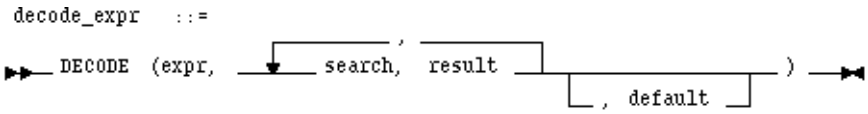

Note that some combinations of functions are inappropriate and are rejected. For example, the LENGTH function is inappropriate within a group function.

**Examples**

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
(TO_CHAR(sysdate,'DD-MMM-YY')
```

***Decoded Expression:***

An expression using the special DECODE syntax.




To evaluate this expression, Personal Oracle Lite compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, Personal Oracle Lite returns the corresponding

*result*. If no match is found, Personal Oracle Lite returns *default*, or, if *default* is omitted, returns null. If *expr* and *search* contain character data, Personal Oracle Lite compares them using non-padded comparison semantics. For information on these semantics, see the section “Datatype Comparison Rules” in *Oracle7 Server SQL Reference*.

The *search*, *result*, and *default* values can be derived from expressions. Personal Oracle Lite evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, Personal Oracle Lite never evaluates a *search* if a previous search is equal to *expr*.

Personal Oracle Lite automatically converts *expr* and each *search* value to the datatype of the first *search* value before comparing. Personal Oracle Lite automatically converts the return value to the same datatype as the first *result*. If the first *result* has the datatype CHAR or if the first *result* is null, then Personal Oracle Lite converts the return value to the datatype VARCHAR2. For information on datatype conversion, see the section “Data Conversion” in *Oracle7 Server SQL Reference*.

In a DECODE expression, Personal Oracle Lite considers two nulls to be equivalent. If *expr* is null, Personal Oracle Lite returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE expression, including *expr*, *searches*, *results*, and *default* is 255.

Refer to the discussion of the CASE function for additional information on the DECODE expression.

**Note:** In Personal Oracle Lite, all arguments in the DECODE expression must be of the same datatype. For example, you cannot decode a numeric column to produce a string, as follows:

```
DECODE (balance, 0, "ZERO BALANCE", balance)
```

**Example**

This expression decodes the value DEPTNO. If DEPTNO is 10, the expression evaluates to ‘ACCOUNTING’; if DEPTNO is 20, it evaluates to ‘RESEARCH’; etc. If DEPTNO is not 10, 20, 30, or 40, the expression returns 'NONE’.

```
DECODE (deptno,10,’ACCOUNTING’,
                              20,     ’RESEARCH’,
                              30,     ’SALES’,
                              40,     ’OPERATION’,
                                      ’NONE’)
```

***List of Expressions:***

A parenthesized list of expressions.


An expression list can contain up to 254 expressions.

**Examples**

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
(LENGTH('MOOSE') * 57, -SQRT(144) + 72, 69)
```

***Usage Notes:***

An *expression* is a combination of one or more values, operators, and SQL functions that evaluates to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype INTEGER (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, resets the time component from the sum to zero, and converts the result to CHAR datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in any of these places:

- the select list of the SELECT command
- a condition of the WHERE and HAVING clauses
- the CONNECT BY, START WITH, and ORDER BY clauses
- the VALUES clause of the INSERT command
- the SET clause of the UPDATE command

For example, you could use an expression in place of the quoted string ‘smith’ in this UPDATE statement SET clause:

```
SET ename = ‘smith’
```

This SET clause has the expression LOWER(ENAME) instead of the quoted string ‘smith’:

```
SET ename = LOWER(ename)
```

### Current Date Predicates

***Description:*** Returns the current date.

***Syntax:***

```
{ fn CURDATE() }                                                [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn CURDATE() *)-- [ODBC]
CURRENT_DATE                                                    [SQL-92]
```

***Key words and parameters:*** None.

***Examples:***

```
{fn CURDATE()}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn CURDATE() *)--
CURRENT_DATE
```

### Current Time Predicates

***Description:*** Returns the current time.

***Syntax:***

```
{ fn CURTIME() }                                                [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn CURTIME() *)-- [ODBC]
CURRENT_TIME                                                    [SQL-92]
```

***Key words and parameters:*** None.

***Comments:*** The time zone of current time is the same as that of the SQL-session.

***Examples:***

```
{fn CURTIME()}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn CURTIME() *)--
CURRENT_TIME
```

### Current Timestamp Predicates

***Description:*** Returns the current timestamp.

***Syntax:***

```
{ fn NOW() }                                                    [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn NOW() *)--[ODBC]
CURRENT_TIMESTAMP                                               [SQL-92]
```

***Key words and parameters:*** None.

***Comments:*** The time zone of current timestamp is the same as that of the SQL-session.

***Examples:***

```
{fn NOW()}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn NOW() *)--
CURRENT_TIMESTAMP
```

### Database Name Predicates

***Description:*** Specifies the database name.

***Syntax:***

```
{ fn DATABASE() }                                              [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn DATABASE() *)--
[ODBC]
```

***Key words and parameters:*** None.

***Comments:*** A database name function returns the same value as that of `SQLGetConnectOption()` with the option `SQL_CURRENT_QUALIFIER`.

***Examples:***

```
{fn DATABASE()}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn DATABASE() *)--
```

### Username Predicates

***Description:*** Specifies name of the user.

***Syntax:***

```
{ fn USER() }                                                  [ODBC]
--(* VENDOR(vendor_name), PRODUCT(product_name) fn USER() *)--[ODBC]
USER                                                           [SQL-92]
```

***Keywords and parameters:*** None.

***Comments:*** A user function returns a user ID if it is supplied during a connection to a data source, and returns an empty string otherwise.

***Examples:***

```
{fn USER()}
--(* VENDOR(vendor_name), PRODUCT(product_name) fn USER() *)--
USER
```

***Related Topics:*** The section “Functions” in *Oracle7 Server SQL Reference*.

The syntax description of ‘*text*'' in *Oracle7 Server SQL Reference*.

The syntax description of *number* in *Oracle7 Server SQL Reference*.

CHAPTER

# 5 Embedded SQL C Binding

This chapter describes the mechanism for embedding SQL statements in a C program and discusses the following types of statements (all unique to embedded SQL):

- embedded SQL C binding
- host variable declaration
- single-row `SELECT`
- cursor-related statements
- dynamic SQL statements
- diagnostics management
- exception declaration
- privilege-related statements

## Embedded SQL C Binding

To compile a C program with embedded SQL statements in Personal Oracle Lite, the program must first be processed by a precompiler application, `esql.exe`. The precompiler translates embedded SQL statements into regular C declarations and statements. The resulting C source code can then be compiled by any C compiler like a regular C program.

C host variables can be used by embedded SQL statements to exchange data with regular C statements. The C data types, which can be used to declare variables for use in an embedded SQL statement, include: `long`, `short`, `float`, `double`, `char`, and `varchar` (`VARCHAR`).

A SQL statement can appear wherever a C statement is allowed. However, each SQL statement must be preceded by the key words `EXEC SQL` and ended with a semicolon. The following example shows how a `CREATE TABLE` statement is embedded in a C program:

```
main()
{
  EXEC SQL CREATE TABLE T1
    (ID INTEGER,NAME CHAR(20),SALARY  DOUBLE);
  /* ... */
}
```

The Embedded SQL precompiler ESQL.EXE processes this source file by interpreting only the syntax between the EXEC SQL statement and the closing semicolon (or between the EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION; statements). ESQL turns the EXEC SQL statements into comments and generates constant character strings, calls to the SQL runtime interfaces and other necessary C code.

## Host and Indicator Variable Declarations

***Description:*** Establishes a declaration section for host variables.

***Syntax:***

```
EXEC SQL BEGIN DECLARE SECTION
/* declaration of C variables used as host variables */
EXEC SQL END DECLARE SECTION
```

***Comments:*** C host variables to be used by embedded SQL statements must first be declared. The C data types for host variables include: `long`, `short`, `float`, `double`, `char`, and `varchar` (`VARCHAR`). Because `varchar` is not a C type, ESQL will translate it into a char type.

Both `char` and `varchar` can be used in an array declaration, in which case the array size must be specified. The size can be a constant, a macro, or a value expression containing both.

When a host variable is used in a SQL statement such as SELECT, it must be preceded by a colon to identify it as a host variable, and not a column. Please see the example in Single-Row `SELECT` Statement.

### Indicator Variables Declaration

Indicator variables must also be declared. They are always of type `Long`. When used in a SQL statement, they must also be preceded by a colon.

Indicator variables have two purposes. When INSERTING or UPDATING, they specify the length of data in a char array, or, if set to -1, they indicate a NULL value to be inserted.

When used in SELECT or FETCH, check the indicator variable. If set to -1, the corresponding column value is NULL. Otherwise, it indicates the length of data in an array, such as char string.

***Example:***

```
EXEC SQL BEGIN DECLARE SECTION;
  short    aShort;
  double   aDouble;
  char     aCharAry[20];
  varchar  aVarcharAry[MAX+1];
  long     ishort, idouble, icharAry; /*indicator variables*/
EXEC SQL END DECLARE SECTION;
```

### Host Variable Specification

Throughout the rest of this chapter, we use `<variable-spec>` to mean a host variable or a host variable paired with an indicator variable.

***Syntax:***

```
: <host variable name>|
: <host variable name> INDICATOR : <indicator variable name>
```

A `<host variable list>` may be a list of such host variables or host and indicator variable pairs separated by commas.

## Single-Row SELECT

***Description:*** Places the resulting values of a `SELECT` statement into a list of host variables.

***Syntax:***

```
SELECT <select_list> INTO <host variable list>
```

```
FROM <table_reference_list>
[ WHERE <search_condition> ]
```

***Key words and parameters:***

*<select_list>*

a list of target columns or expressions. See Select Statement.

*<host_variable_list>*

a list of *<variable_spec>* host variable specifications separated by commas, one for each item in the `select_list`.

If a selected value is NULL, the corresponding indicator variable is set to -1.

*<table_reference_list>*

a list of source tables. See Select Statement.

*<search_condition>*

a search condition. See Search Condition.

***Comments:*** The resulting set of the `SELECT` statement should contain only one row. More than one row generates an error.

An indicator variable must be of the type `long`.

***Example:***

```
EXEC SQL BEGIN DECLARE SECTION;
  short   EmpID;
  long    Ind;
  double      AvgSalary;
  char        EmpName[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT ID, NAME
  INTO :EmpID INDICATOR :Ind, :EmpName
  FROM EMPLOYEE
  WHERE SALARY >= :AvgSalary;
```

## Cursor-related Statements

This section describes statements for declaring, opening, fetching, and closing cursors. Cursors are a mechanism for managing multiple row result sets. The statements described in this section operate on fixed cursors which are associated with a single SQL statement, provided at compile time.

### DECLARE CURSOR

***Description:*** Declares a cursor on a `SELECT` statement for future use.

***Syntax:***

```
DECLARE <cursor_name> CURSOR FOR <cursor_specification>
```

***Key words and parameters:***

*<cursor_name>*
name of the cursor (an identifier).

*<cursor_specification>*
a SELECT statement.

***Comments:*** In many cases, the result of a SELECT statement contains multiple rows. SQL introduces the notion of cursor to enable a host program to retrieve these rows one at a time. The DECLARE CURSOR statement registers such a cursor for future use. It does not result in the actual execution of the SELECT statement.

***Example:***

```
EXEC SQL DECLARE cursor1 CURSOR FOR
  SELECT ID, NAME FROM EMPLOYEE;
```

## OPEN

***Description:*** Opens a previously declared cursor.

***Syntax:*** `OPEN <cursor_name>`

***Key words and parameters:***

*<cursor_name>*
name of the cursor (an identifier).

***Comments:*** An OPEN statement must be textually preceded by a DECLARE CURSOR statement on the cursor. If the cursor is not in the closed state, an exception condition is raised: invalid cursor state.

***Example:***

```
EXEC SQL OPEN cursor1;
```

## FETCH

***Description:*** Retrieves data through an open cursor.

***Syntax:*** `FETCH <cursor_name> INTO <host_variable_list>`

***Key words and parameters:***

*<cursor_name>*
name of the cursor (an identifier).

*<host_variable_list>*
a list of host variables, separated by commas.

***Comments:*** A FETCH statement must be textually preceded by a DECLARE CURSOR statement on the cursor. If the cursor is not in the open state, an exception condition is raised: invalid cursor state. There should be one host variable for each column in the select list.

***Example:***

```
EXEC SQL FETCH cursor1 INTO :EmpID, :EmpName;
```

### CLOSE

***Description:*** Closes an open cursor.

***Syntax:*** `CLOSE <cursor_name>`

***Key words and parameters:***

*`<cursor_name>`*
name of the cursor (an identifier).

***Comments:*** A `CLOSE` statement must be textually preceded by a `DECLARE CURSOR` statement on the cursor. If the cursor is not in the open state, an exception condition is raised: invalid cursor state.

***Example:***

```
EXEC SQL CLOSE cursor1;
```

## Dynamic SQL Statements

Dynamic cursors associate a cursor name in a string variable with a SQL statement that may be generated by the program or by user input. The SQL statement need not be known at compile time. A cursor can be closed and reused for more than one SQL statement.

### DYNAMIC DECLARE CURSOR

***Description:*** Dynamically associates a cursor with a statement.

***Syntax:*** `DECLARE <cursor_name> CURSOR FOR <statement_name>`

***Key words and parameters:***

*`<cursor_name>`*
name of the cursor (an identifier). See `DECLARE CURSOR` Statement.

*`<statement_name>`*
name of the statement (an identifier).

***Comments:*** The *`<statement_name>`* represents a prepared statement, which contains a *`<cursor_specification>`* as in `DECLARE CURSOR` Statement. Cursor names and statement names do not have colons preceding them.

***Examples:***

```
EXEC SQL DECLARE Cursor1 CURSOR FOR Stmt1;
```

### ALLOCATE CURSOR

***Description:*** Allocates a dynamic cursor (given a host variable containing the cursor name) for a statement (given a host variable containing the statement name).

***Syntax:***

```
ALLOCATE <extended_cursor_name> CURSOR
  FOR <extended_statement_name>
```

***Key words and parameters:***

*<extended_cursor_name>*
a host variable whose value represents a cursor name.

*<extended_statement_name>*
a host variable whose value represents a statement name.

***Comments:*** The statement indicated by *<extended_statement_name>* must be a prepared statement.

***Examples:***

```
EXEC SQL ALLOCATE :Cursor1 CURSOR FOR :Stmt1;
```

## DYNAMIC OPEN

***Description:*** Opens a dynamic cursor and optionally binds input parameters to its associated statement.

***Syntax:***

```
OPEN <dynamic_cursor_name> [ USING <host_variable_list> ]
```

***Key words and parameters:***

*<dynamic_cursor_ name>*
a <cursor_name> as in DYNAMIC DECLARE statement, or an <extended cursor name> as in ALLOCATE CURSOR Statement.

*<host_variable_ list>*
a list of host variables separated by commas. See EXECUTE Statement.

***Comments:*** The *<cursor_name>* identifies a cursor already declared. The statement name associated with the cursor identifies a prepared statement.

The *<extended_cursor_name>* identifies a cursor already allocated.

***Example:***

```
EXEC SQL OPEN :Cursor1;
EXEC SQL OPEN Cursor1 USING :AvgSalary;
```

## DYNAMIC FETCH

***Description:*** Fetches a row through a dynamic cursor declared with a DYNAMIC DECLARE CURSOR statement.

***Syntax:***

```
FETCH <dynamic_cursor_name>
  [ INTO <host_variable_list> ]
```

***Key words and parameters:***

*<dynamic_cursor_name>*

see DYNAMIC OPEN Statement.

*<host_variable_list>*

a list of host variables separated by commas.

***Examples:***

```
EXEC SQL FETCH :Cursor1 INTO :EmpID, :EmpName;
```

## DYNAMIC CLOSE

***Description:*** Close a dynamic cursor.

***Syntax:*** `CLOSE <dynamic_cursor_name>`

***Key words and parameters:***

*<dynamic_cursor_name>*

See DYNAMIC OPEN Statement.

***Examples:***

```
EXEC SQL CLOSE :Cursor1;

EXEC SQL CLOSE Cursor1;
```

## PREPARE

***Description:*** Prepares a statement for execution.

***Syntax:*** `PREPARE <sql_statement_name> FROM <sql_statement_variable>`

***Key words and parameters:***

*<sql_statement_name>*

the name for the prepared statement, either an <identifier> or a host variable.

*<sql_statement_ variable>*

a host variable containing the SQL statement to be prepared.

***Comments:*** Dynamic SQL enables the execution of SQL statements at run time without prior knowledge of the statements at compile time. As such, a programmer does not need to hard-code those SQL statements into an embedded SQL/C program. Instead, these statements can be formulated and prepared at run time through host variables.

Most SQL statements are preparable. Some exceptions include:

- dynamic SQL statements
- single-row SELECT statement
- cursor statements (either static or dynamic)

A preparable statement can not contain any host variables. Instead, use the dynamic parameter markers when necessary. A dynamic parameter marker is a question mark "?" whose value is filled in during execution through a host variable identified in the USING clause of an EXECUTE statement. See the EXECUTE Statement for details.

***Example:***

```
EXEC SQL BEGIN DECLARE SECTION;
  short EmpID;
  double AvgSalary;
  char EmpName[20];
  char QryStr1[128] =
  "SQL SELECT ID,NAME FROM EMPLOYEE WHERE SALARY > ?";
  char QryStr2[128] =
  "SQL SELECT ID, NAME FROM EMPLOYEE";
  char Stmt1[8] = "Stmt1";
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE :Stmt1 FROM :QryStr1;
EXEC SQL PREPARE Stmt2 FROM :QryStr2;
```

Note that the first statement prepares a SQL statement and associates it with the identifier Stmt1 contained in the host variable Stmt1, while the second statement associates another SQL statement directly with the SQL identifier Stmt2.

## EXECUTE

***Description:*** Binds the input parameters and output targets to a prepared statement and executes the statement.

***Syntax:***

```
EXECUTE <sql_statement_name>
  [ INTO <host_variable_list> ]
  [ USING <host_variable_list> ]
```

***Key words and parameters:***

*<sql_statement_name>*
the name for the prepared statement. See PREPARE Statement.

*<host_variable_list>*
a list of host variables separated by commas.

***Examples:***

```
/* extending from example in Prepare Statement ... */
EXEC SQL EXECUTE :Stmt1
  INTO :EmpID, :EmpName
  USING :AvgSalary;
EXEC SQL EXECUTE Stmt2;
```

## EXECUTE IMMEDIATE

***Description:*** Dynamically prepares and executes a preparable statement.

***Syntax:***

```
EXECUTE IMMEDIATE <sql_statement_variable>
```

***Key words and parameters:***

*<sql_statement_variable>*

a host variable containing the SQL statement to be prepared.

***Comments:*** This statement has the same effect as the PREPARE and EXECUTE statements combined. Note that there is no way to pass in host variables for execution. Therefore, the SQL statement must not contain any dynamic parameter markers.

***Examples:***

```
EXEC SQL BEGIN DECLARE SECTION;
  char QryStr[128] =
    "SQL SELECT ID, NAME FROM EMPLOYEE";
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE IMMEDIATE :QryStr;
```

## Diagnostics Management

***Description:*** Returns exception or completion information from the SQL diagnostics area.

***Syntax:***

```
GET DIAGNOSTICS
GET DIAGNOSTICS <statement_information>
GET DIAGNOSTICS <condition_information>
_____________________________________________
<statement_information> ::=
  <statement_information_item>
    [ { , <statement_information_item> }... ]
<statement_information_item> ::=
<embedded_variable_name> = <statement_information_item_name>
<condition_information> ::=
  EXCEPTION <unsigned_integer>
  <condition_information_item>
    [ { , <condition_information_item> }... ]
<condition_information_item> ::=
<embedded_variable_name> = <condition_information_item_name>
```

***Key words and parameters:***

*<statement_information>*

the information from the header area.

*<embedded_variable_name>*

the name of a host variable prefixed by ‘: ‘.

*<statement_ information_item_name>*

one of the following items: NUMBER, MORE, COMMAND_FUNCTION, DYNAMIC_FUNCTION, or ROW_COUNT.

*<condition_information>*

the information from the detailed area.

*<condition_ information_item_name>*

one of the following items:

| | | |
|---|---|---|
| CONDITION_NUMBER | CONNECTION_NAME | CURSOR_NAME |
| RETURNED_SQLSTATE | CATALOG_NAME, SCHEMA_NAME | MESSAGE_TEXT |
| CLASS_ORIGIN | TABLE_NAME | MESSAGE_LENGTH |
| SUBCLASS_ORIGIN | COLUMN_NAME | MESSAGE_OCTET_ LENGTH |
| SERVER_NAME | | |

***Comments:*** The execution condition of a SQL statement can be obtained from a SQL diagnostics area maintained by the database system. The diagnostics area is comprised of two components. The first component is the header area, which contains general information about the last executed SQL statement. The second area is a detailed area, which contains information about a specific error, warning, or success code.

For simpler diagnostic usage, Embedded SQL specifies that `SQLCODE` and `SQLSTATE` be used as two host variables indicating the execution condition of a SQL statement. Users are expected to declare either or both statements in their host program. However, if none is declared, the precompiler declares `SQLCODE` for the user.

`SQLCODE` must be defined as long type. `SQLSTATE` must be defined as char[6]. If they are not defined exactly this way, their presence is not recognized by the precompiler and hence their use can lead to compilation errors reported by the C compiler.

The values of `SQLCODE` and `SQLSTATE` are set by the database system after the execution of each SQL statement. Possible values for `SQLCODE` are:

| **`SQLCODE` Value** | **Condition** |
|---|---|
| 0 | successful completion |
| 100 | no data |
| -n | exception |

The five characters in a `SQLSTATE` are logically divided into two components. The first two are called the *class code*, and the last three characters are called the *subclass code*. Any class code that begins with the characters A through H or 0 through 4 indicates a `SQLSTATE` that is defined by the standard. Any class code starting with the characters I through Z or 5 through 9 is implementation-defined. Personal Oracle Lite

SQL currently does not have any implementation-defined values for SQLSTATE, except those added by the ODBC standard.

***Examples:***

```
EXEC SQL GET DIAGNOSTICS
  :Num = NUMBER, :CmdFn = COMMAND_FUNCTION;
EXEC SQL GET DIAGNOSTICS EXCEPTION 1
  :Cond = CONNECTION_NAME, :Cond = CURSOR_NAME;
```

## Exception Declaration

***Description:*** Designates an action to be taken when certain situations occur.

***Syntax:***

```
WHENEVER <condition> CONTINUE
WHENEVER <condition> GOTO <label>
```

***Key words and parameters:***

*<condition>*
execution result of a SQL statement. It can be SQLERROR or NOT FOUND.

*<label>*
a C program label.

***Comments:*** This statement establishes an appropriate action for all embedded SQL statements that textually follow it in an embedded SQL program. Another WHENEVER statement sets a new course of action for all subsequent embedded SQL statements. This statement alleviates the need for writing error handling code after each embedded SQL statement.

***Examples:*** The following example specifies that the program continue running when the execution of a SQL statement returns an error:

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
```

The following example specifies that the program control flow be transferred to label Cleanup if the execution of a SQL statement (such as a FETCH statement) results in no data found.

```
EXEC SQL WHENEVER NOT FOUND GOTO Cleanup;
```

## Privilege-related Statements

### GRANT Statement

***Description:*** Defines privileges on tables.

***Syntax:***

```
GRANT ALL ON <table_name> TO <user_list>
GRANT <privileges> ON <table_name> TO <user_list>
```

***Key word s and parameters:***

*<table_name>*
the name of the table on which privileges are granted.

*<column_commalist>*
a list of column names separated by commas.

***Comments:*** Privilege information is recorded but not enforced in this release.

***Example:***

```
EXEC SQL GRANT SELECT, REFERENCES ON Employee TO PUBLIC;
EXEC SQL GRANT ALL ON Employee TO Personnel, Executive;
```

### REVOKE

***Description:*** Remove privileges of users on tables.

***Syntax:***

```
REVOKE ALL ON <table_name> FROM <user_list>
EXEC SQL REVOKE <privileges> ON <table_name> FROM <user_list>;
```

***Keywords and parameters:***

*<table_name>*
the name of the table on which privileges are granted.

*<privileges>*
See GRANT Statement.

*<user_list>*
See GRANT Statement.

***Comments:*** Privilege information is recorded but not enforced in this release.

***Example:***

```
REVOKE INSERT, DELETE, UPDATE ON Employee FROM PUBLIC
```

CHAPTER

# 6 ODBC Binding

This chapter describes:

- ODBC binding
- key words, syntax and examples

## ODBC Binding

The Open Database Connectivity (ODBC) interface is a Call Level Interface (CLI). ODBC consists of a library of functions that support SQL statements. An application using SQL as its query language can be programmed against the ODBC interface to access objects in a database. The interoperability supported by ODBC is achieved through the notion of *driver* and *data source*. Objects in a data source are accessed and manipulated through a driver program that interprets a set of standard requests (in terms of ODBC functions) for the specific data source. As long as an application adheres to the ODBC interface specification, interoperability in accessing different data sources is guaranteed.

The Personal Oracle Lite ODBC (OOT ODBC for short) driver is one such driver that allows an application to access the Personal Oracle Lite database by calling functions in the ODBC interface. The OOT ODBC driver is of single-tier type; it processes and passes SQL statements to the Personal Oracle Lite database for processing. This chapter describes all the ODBC functions supported by the OOT ODBC driver.

**Note:** The items in uppercase are typedefs defined in SQLTYPES.H from the Microsoft ODBC Software Developer's Kit. For example, all of the functions return a result of type RETCODE which is defined as a signed short.

### `SQLAllocConnect`

***Description:*** Allocates memory for a connection handle within an environment.

***Syntax:***

```
RETCODE SQLAllocConnect(HENV henv, HDBC *phdbc);
```

***Key words and parameters:***

`henv`
an environment handle.

`phdbc`
a pointer to the buffer for the connection handle.

***Comments:*** An application calls this function by passing an environment handle and the buffer address of a connection handle. A connection handle is needed for calls to `SQLAllocStmt`, as well as some other functions. This function is directly handled by the driver manager, in that the `SQLAllocConnect` function in the driver is actually called later when the application makes a call to `SQLConnect` or `SQLDriverConnect`.

***Examples:*** See `SQLConnect`.

### SQLAllocEnv

***Description:*** Allocates memory for an environment handle.

***Syntax:***
```
RETCODE SQLAllocEnv(HENV *phenv);
```

***Key words and parameters:***

`phenv`
a pointer to the buffer for the environment handle.

***Comments:*** An application calls this function to set up an environment for making ODBC connections to the Personal Oracle Lite SQL server. Before calling any other ODBC function, an application must call this function to set up an environment. An application can have one environment allocated at any one time.

***Examples:*** See `SQLConnect`.

### SQLAllocStmt

***Description:*** Allocates memory for a statement handle and associates it with a given connection handle.

***Syntax:***
```
RETCODE SQLAllocStmt(HDBC hdbc, HSTMT *phstmt);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`phstmt`
a pointer to the buffer for the statement handle.

***Comments:*** A statement handle references statement information, which is needed for access to the target Personal Oracle Lite SQL server. An application must call this function to allocate a statement handle before issuing a SQL statement. This function allocates memory for the statement information and returns a pointer to the handle. Most other ODBC functions require a statement handle as an input argument to provide the necessary context for a SQL statement.

***Examples:*** See `SQLConnect`.

### SQLBindCol

***Description:*** Assigns the buffer and data type for accessing a column in a result set.

***Syntax:***
```
RETCODE SQLBindCol(HSTMT hstsmt, UWORD icol, SWORD fCType,
      PTR rgbValue, SDWORD cbValueMax, SDWORD *pcbValue);
```

***Key words and parameters:***

`hstmt`

a statement handle.

`icol`

a column position in the result set, ordered sequentially from left to right, starting at 1.

`fCType`

a C type indicator for the receiving buffer. The OOT ODBC driver accepts the following data types: SQL_C_BINARY, SQL_C_BIT, SQL_C_CHAR, SQL_C_DATE, SQL_C_DEFAULT, SQL_C_DOUBLE, SQL_C_FLOAT, SQL_C_LONG, SQL_C_SHORT, SQL_C_SLONG, SQL_C_SSHORT, SQL_C_STINYINT, SQL_C_TIME, SQL_C_TIMESTAMP, SQL_C_TINYINT, SQL_C_ULONG, SQL_C_USHORT, SQL_C_UTINYINT.

`rgbValue`

a pointer to a buffer for receiving the data of a column in a result set. If rgbValue is a null pointer, the driver unbinds the column.

`cbValueMax`

the maximum length of the rgbValue buffer. For character data, this number includes the space for a null terminator.

`pcbValue`

a pointer to a buffer for returning the actual length of data in rgbValue. For character data, the length does not include the null terminator. When a NULL value is retrieved, this buffer is set to SQL_NULL_DATA.

***Comments:***

The ODBC interface provides two ways to retrieve the data of a column: `SQLBind-Col` and `SQLGetData`. The latter is an extended function to be documented later.

This function informs the driver of:

- the data type of the column data the application expects to receive
- a buffer for receiving the column data
- the length of the receiving buffer
- a buffer location for receiving the actual length of the data

Subsequently, when the application calls the `SQLFetch` function, the data in the column is placed into the buffer specified. Before each subsequent call to the `SQLFetch` function, an application can freely call `SQLBindCol` to change the buffer location and data type of the buffer. The application is responsible for making sure that the buffer is large enough to hold the data to be received.

***Examples:***

```
#define NAME_LENGTH 20
#define MESSAGE_LENGTH60
#define STATE_LENGTH40
HENV  henv; /* environment handle */
HDBC  hdbc;/* connection handle */
HSTMT hstmt;/* statement handle */
RETCODE   rc;   /* ODBC function return code */
UCHAR szFirstName[NAME_LENGTH];
UCHAR szLastName[NAME_LENGTH];
SWORD sId;
SDWORD  cbFirstName, cbLastName, cbId;
UCHAR   szSqlState[STATE_LENGTH];
SDWORD  pfNativeError;
UCHAR   szErrorMsg[MESSAGE_LENGTH];
SWORD   pcbErrorMsg;

rc = SQLExecDirect(hstmt,
  "SELECT ID,FIRST_NAME,LAST_NAME FROM EMPLOYEE", SQL_NTS);
if (rc == SQL_SUCCESS) {
  /* Bind columns 1, 2, and 3 */
  SQLBindCol(hstmt, 1, SQL_C_SSHORT, &sId, 0, &cbId);
  SQLBindCol(hstmt, 2, SQL_C_CHAR,
        szFirstName, NAME_LENGTH, &cbFirstName);
  SQLBindCol(hstmt, 3, SQL_C_CHAR,
        szLastName, NAME_LENGTH, &cbLastName);

  /* fetch each row and display an error message if an
    error occurred */
  while (TRUE) {
    rc = SQLFetch(hstmt);/* get next row */
    if (rc == SQL_ERROR || rc == SQL_SUCCESS_WITH_INFO) {
      SQLError(henv, hdbc, hstmt, szSqlState,
          &pfNativeError, szErrorMsg,
          MAX_MESSAGE_LEN, &pcbErrorMsg);
```

```
      /* handle the error ... */
    }
    if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO){
      /* process the row ... */
    } else
      break;
  }
}
```

## SQLBindParameter

***Description:*** Binds a buffer to a parameter marker in a SQL statement for passing input parameter values.

***Syntax:***

```
RETCODE SQLBindParameter(HSTMT hstmt, UWORD ipar,
    SWORD fParamType, SWORD fCType, SWORD fSqlType,
    UDWORD cbColDef, SWORD ibScale, PTR rgbValue,
    SDWORD cbValueMax, SDWORD *pcbValue);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`ipar`
a parameter position, ordered sequentially from left to right, starting from 1.

`fParamType`
a type indicator for the parameter.

`fCType`
the C data type of the parameter. The OOT ODBC driver accepts the following data types: SQL_C_BINARY, SQL_C_BIT, SQL_C_CHAR, SQL_C_DATE, SQL_C_DEFAULT, SQL_C_DOUBLE, SQL_C_FLOAT, SQL_C_LONG, SQL_C_SHORT, SQL_C_SLONG, SQL_C_SSHORT, SQL_C_STINYINT, SQL_C_TIME, SQL_C_TIMESTAMP, SQL_C_TINYINT, SQL_C_ULONG, SQL_C_USHORT, SQL_C_UTINYINT.

`fSqlType`
the SQL data type of the parameter. The OOT ODBC driver accepts the following SQL data types: SQL_BIGINT, SQL_BINARY, SQL_BIT, SQL_CHAR, SQL_DATE, SQL_DECIMAL, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_LONGVARCHAR, SQL_NUMERIC, SQL_REAL, SQL_SAMLLINT, SQL_TIME, SQL_TIMESTAMP, SQL_TINYINT, SQL_VARBINARY, and SQL_VARCHAR.

`cbColDef`
the precision of the column or expression of the corresponding parameter marker.

`ibScale`
the scale of the column or expression of the corresponding parameter marker.

`rgbValue`
a pointer to a buffer for the parameter's data.

`cbValueMax`
the maximum length of the rgbValue buffer.

`pcbValue`
a pointer to a buffer for returning the length of the parameter's data.

***Comments:*** An application calls `SQLBindParameter` to bind each parameter marker in an SQL statement to an input data buffer. The bindings remain in effect until the application calls `SQLBindParameter` again to change the binding, or calls `SQLFreeStmt` with the `SQL_DROP` or `SQL_RESET_PARAMS` option. The actual input values are bound to the parameter markers in a SQL statement right before the driver passes the request to the Personal Oracle Lite SQL server for processing. There are no immediate effects after calling this function. As such, the same buffer must not be used to bind different data values for different parameter markers in the same SQL statement.

***Examples:***

```
#define NAME_LENGTH 30
UCHAR szFirstName[NAME_LENGTH];
UCHAR szLastName[NAME_LENGTH];
SWORD sId;
SDWORD  cbId = 0;
SDWORD  cbFirstName = SQL_NTS;
SDWORD  cbLastName = SQL_NTS;
RETCODE   rc;   /* ODBC function return code */

rc = SQLPrepare(hstmt, "INSERT INTO EMPLOYEE (ID,\
  FIRST_NAME, LAST_NAME) VALUES (?, ?, ?)", SQL_NTS);
if (rc == SQL_SUCCESS) {
  /* specify types & buffers for ID, FIRST_NAME & LAST_NAME */
  SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SSHORT,
    SQL_SMALLINT, 0, 0, &sId, 0, &cbId);
  SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
    SQL_CHAR, NAME_LENGTH, 0, szFirstName, 0, &cbFirstName);
  SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR,
    SQL_CHAR, NAME_LENGTH, 0, szLastName, 0, &cbLastName);
  /* specify data for the first row & insert */
  sId = 10;
  strcpy(szFirstName, "Doris");
  strcpy(szLastName, "Saye");
  rc = SQLExecute(hstmt);
```

```
  /* Specify data for subsequent rows & execute ... */
}
```

## SQLBrowseConnect

***Description:*** SQLBrowseConnect supports an iterative method of listing the attributes and attribute values required for connection. Each call to SQLBrowseConnect with a partial connection string returns the next level of attributes, and a return code of SQL_NEED_DATA. When all the attributes are specified, the driver connects to the data source and returns SQL_SUCCESS and a complete connection string.

***Syntax:***

```
RETCODE SQLBrowseConnect(HDBC hdbc,
    UCHAR FAR *szConnStrIn, SWORD cbConnStrIn,
    UCHAR FAR *szConnStrOut, SWORD cbConnStrOutMax,
    SWORD *pcbConnStrOut);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`szConnStrIn`
an input connection string.

`cbConnStrIn`
the length of szConnStrIn.

`szConnStrOut`
a pointer to a buffer for the browse result connection string.

`cbConnStrOutMax`
the maximum length of the szConnStrOut buffer.

`pcbConnStrOut`
a pointer to the total number of bytes (excluding the null termination byte) available for return in szConnStrOut. If the number of bytes available for return is greater than or equal to cbConnStrOutMax, the completed connection string in szConnStrOut is truncated to cbConnStrOutMax-1 bytes.

***Comments:*** A browse request connection string szConnStrIn has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= attribute-keyword=attribute-value | DRIVER={attribute-value}
(The braces are literal; the application must specify them.)

attribute-keyword ::= DSN | UID | PWD| DataDirectory

attribute-value ::= character-string

driver-defined-attribute-keyword ::= identifier

where a character-string contains zero or more characters and an identifier contains

one or more characters.

The browse result connection string (szConnStrOut) is a list of connection attributes. An attribute consists of a keyword and a value. The browse result connection string has the following syntax:

connection-string ::= attribute[;] | attribute; connection-string

attribute ::= [*]attribute-keyword=attribute-value

attribute-keyword ::= ODBC-attribute-keyword| DataDirectory

ODBC-attribute-keyword = { DSN | UID | PWD }

attribute-value ::= {attribute-value-list} | ?

attribute-value-list ::= character-string | character-string, attribute-value-list

where a character-string contains zero or more characters and an identifier contains one or more characters.

***Examples:***

```
HDBC  hdbc; /* connection handle */
SDWORD  NumBytes;/* bytes in a return value */
char  BrowseResult[MAX_CONN_STR];

BrowseResult[0] = '\0';
rc = SQLBrowseConnect(hdbc, "DSN=OOT", SQL_NTS,
    BrowseResult, MAX_CONN_STR, &NumBytes);
if(rc == SQL_NEED_DATA){
  rc = SQLBrowseConnect(hdbc, "UID=DORIS;PWD=DORIS;",
      SQL_NTS, BrowseResult, MAX_CONN_STR, &NumBytes);
  if(rc != SQL_SUCCESS) { /* error handling */ }
}
```

## SQLCancel

***Description:*** Cancels the processing of a SQL statement.

***Syntax:***

```
RETCODE SQLCancel(HSTMT hstmt)
```

***Key words and parameters:***

`hstmt`
a statement handle.

***Comments:*** Personal Oracle Lite SQL server does not support asynchronous execution. The effect

of this function is equivalent to the SQLFreeStmt with SQL_CLOSE option.

***Examples:***

```
HSTMT hstmt;/* statement handle*/
RETCODE  rc;   /* ODBC function return code */
  rc = SQLExecute(hstmt);
  /* If SQLExecute returns SQL_NEED_DATA and user
     wishes to cancel the execution, then call SQLCancel*/
  if (rc == SQL_NEED_DATA) {
    /* prompt user for choice */
    cancel = GetUserChoice();
    if (cancel)
      rc = SQLCancel(hstmt);
    else {
      /* continue execution ... */
    }
  }
}
```

## SQLColAttributes

***Description:*** Returns descriptor information for a column in a result set.

***Syntax:***

```
RETCODE SQLColAttributes(HSTMT hstmt, UWORD icol,
    UWORD fDescType, PTR rgbDesc, SWORD cbDescMax,
    SWORD *pcbDesc, SDWORD *pfDesc);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`icol`
a column position in a result set, ordered sequentially from left to right, starting at 1.

`fDescType`
a descriptor type.

`rgbDesc`
a pointer to a buffer for receiving descriptor information. The format of the descriptor information returned depends on fDescType.

`cbDescMax`
the length of the rgbDesc buffer.

`pcbDesc`
the actual number of bytes (excluding the null termination byte for character data) returned in rgbValue. For character data, if the number of bytes available is greater than or equal to cbDescMax, the descriptor information in rgbDesc is truncated to cbDescMax-1 bytes and is null-terminated by the driver.

`pfDesc`

a pointer to an integer value for receiving descriptor information of numeric descriptor types, such as SQL_COLUMN_LENGTH.

***Comments:*** `SQLColAttributes` is an extended alternative to `SQLDescribeCol`. `SQLDescribeCol` returns a fixed set of descriptor information based on the ANSI-89 SQL standard. `SQLColAttributes` allows access to a more extensive set of descriptor information available in ANSI-92. The descriptor information returned is a character string, a 32-bit descriptor-dependent value, or a 16-bit integer value. An application can access descriptor information in any order of column position.

***Examples:***

```
#define NAME_LEN 30
HSTMT hstmt;   /* statement handle */
char  szColName[NAME_LEN];/* column name buffer */
SDWORD  fDesc;   /* column type code */
SDWORD  fDesc1;  /* column precision */

SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL_NTS);
/* extract information from column 1 of result set */
SQLColAttributes(hstmt, 1, SQL_COLUMN_NAME, NULL,
         szColName, &cbDesc, NULL);
/* print name of column 1 in szColName ... */
SQLColAttributes(hstmt, 1, SQL_COLUMN_TYPE, NULL, 0, NULL, &fDesc);
/* print type code of column 1 in fDesc ... */
SQLColAttributes(hstmt, 1, SQL_COLUMN_PRECISION, NULL,
       0, NULL, &fDesc1);
/* print precision of column 1 in fDesc1 ... */
```

## SQLColumnPrivileges

***Description:*** Returns as a result set the information about the privileges associated with the columns of the specified tables.

***Syntax:***

```
RETCODE SQLTablePrivileges(HSTMT hstmt,
    UCHAR FAR *szTableQualifier, SWORD cbTableQualifier,
    UCHAR FAR *szTableOwner, SWORD cbTableOwner,
    UCHAR FAR *szTableName, SWORD cbTableName,
    UCHAR FAR *szColumnName, SWORD cbColumnName);
```

***Key words and parameters:***

`hstmt`
a statement handle.
`szTableQualifier`
a qualifier name.
`cbTableQualifier`
the length of szTableQualifier.
`szTableOwner`
a string search pattern for owner names.
`cbTableOwner`
the length of szTableOwner.
`szTableName`
a string search pattern for table names.
`cbTableName`
the length of szTableName.
`szColumnName`
a string search pattern for column names.
`cbColumnName`
the length of szColumnName.

***Comments:***

SQLColumnPrivileges returns the information as a result set, ordered by TABLE_QUALIFIER, TABLE_OWNER, TABLE_NAME, COLUMN_NAME, and PRIVILEGE.

The result set consists of the following columns:

| Column Name | Data Type | Comments |
|---|---|---|
| TABLE_QUALIFIER | Varchar(128) | table qualifier identifier |
| TABLE_OWNER | Varchar(128) | table owner identifier |
| TABLE_NAME | Varchar(128) not NULL | table identifier |
| COLUMN_NAME | Varchar(128) not NULL | column identifier |
| GRANTOR | Varchar(128) | identifier of the user who granted the privilege |
| GRANTEE | Varchar(128) not NULL | identifier of the user to whom the privilege was granted |
| PRIVILEGE | Varchar(128) not NULL | identifies the table privilege |
| IS_GRANTABLE | Varchar(3) | indicates whether the grantee is permitted to grant the privilege to other users; "YES", "NO", or NULL if unknown or not applicable to the data source. |

***Examples:***

```
HSTMT hstmt;/* statement handle */
SDWORD  NumBytes;/* bytes in a return value */
UCHAR TableQualifier[MAX_NAME_LEN];
UCHAR TableOwner[MAX_NAME_LEN];
UCHAR TableName[MAX_NAME_LEN];
UCHAR ColumnName[MAX_NAME_LEN];
UCHAR Grantor[MAX_NAME_LEN];
UCHAR Grantee[MAX_NAME_LEN];
UCHAR Privilege[MAX_NAME_LEN];
UCHAR IsGrantable[MAX_NAME_LEN];
RETCODE rc;   /* ODBC function return code */

rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
  TableQualifier, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
  TableOwner, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 3, SQL_C_CHAR,
  TableName, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 4, SQL_C_CHAR,
  ColumnName, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 5, SQL_C_CHAR,
  Grantor, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
  Grantee, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 7, SQL_C_CHAR,
  Privilege, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 8, SQL_C_CHAR,
  IsGrantable, MAX_NAME_LEN, &NumRetBytes);

rc = SQLColumnPrivileges(hstmt, "POLITE", SQL_NTS,
  "OOT_SCH", SQL_NTS, "EMPLOYEE", SQL_NTS, "NAME", SQL_NTS);
for(;;) {
  rc = SQLFetch(hstmt);
  if(rc != SQL_SUCCESS) break;
```

```
  /* process the information */
} /* for */
```

## SQLColumns

***Description:*** Returns the list of column names in specified tables.

***Syntax:***

```
RETCODE SQLColumns(HSTMT hstmt,
    UCHAR szTableQualifier, SWORD cbTableQualifier,
    UCHAR *szTaleOwner, SWORD cbTableOwner,
    UCHAR *szTableName, SWORD cbTableName,
    UCHAR *szColumnName, SWORD cbColumnName)
```

***Key words and parameters:***

`hstmt`
a statement handle.

`szTableQualifier`
a table qualifier name.

`cbTableQualifier`
the length of szTableQualifier.

`szTableOwner`
a string search pattern for owner names.

`cbTableOwner`
the length of szTableOwner.

`szTableName`
a string search pattern for table names.

`cbTableName`
the length of szTableName.

`szColumName`
a string search pattern for column names.

`cbColumnName`
the length of szColumnName.

***Comments:*** This function returns the column information as a standard result set on the given statement handle. An application needs to use either `SQLFetch` or `SQLGetData` to access the results. The rows in the results are ordered sequentially by the table name only.

This function is typically called before statement execution to retrieve information about columns of a table or tables from the Personal Oracle Lite database. By contrast, `SQLDescribeCol` describes the columns in a result set, and `SQLNumResultCols` returns the number of columns in a result set. `SQLColumns` does not describe computed columns in a fetch, nor does it return the number of

columns referenced by a `SELECT` statement.

The result set returned by `SQLColumns` contains the following columns:

| Column Name | Data Type | Comments |
|---|---|---|
| TABLE_QUALIFIER | VARCHAR(128) | The catalog name, defaulting to `POLITE`. |
| TABLE_OWNER | VARCHAR(128) | Table owner identifier, defaulting to `OOT_SCH`. |
| TABLE_NAME | VARCHAR(128) | Table identifier. |
| COLUMN_NAME | VARCHAR(128) | Column identifier. |
| DATA_TYPE | SMALLINT | ODBC SQL data type. |
| TYPE_NAME | VARCHAR(128) | SQL type name. |
| PRECISIONS | INTEGER | Column precision. |
| LENGTHS | INTEGER | The length in bytes of data transferred on an `SQLGetData` or `SQLFetch` operation if `SQL_C_DEFAULT` is specified. |
| SCALE | SMALLINT | The scale of the column. |
| RADIX | SMALLINT | 10 if the column is of a numeric data type; `NULL` if radix is not applicable to the column data type. |
| NULLABLE | SMALLINT | Column nullable indicator: `SQL_NO_NULLS` if the column does not accept `NULL` values; `SQL_NULLABLE` if the column accepts `NULL` values; `SQL_NULLABLE_UNKNOWN` if it is not known if the column accepts `NULL` values. |
| REMARKS | VARCHAR(128) | Table description. Not available in the current release and is always `NULL`. |

***Examples:***

```
#define STR_LEN(128+1)
#define REM_LEN (254+1)
/* Declare buffer for the result set data */
UCHAR szQualifier[STR_LEN], szOwner[STR_LEN];
UCHAR szTableName[STR_LEN], szColName[STR_LEN];
UCHAR szTypeName[STR_LEN], szRemarks[REM_LEN];
SDWORD  Precision, Length;
SWORD DataType, Scale, Radix, Nullable;
RETCODErc; /* ODBC function return code */
/* Declare buffer for the actual length of data */
SDWORD  cbQualifier, cbOwner, cbTableName, cbColName;
SDWORD  cbTypeName, cbRemarks, cbDataType, cbPrecision;
SDWORD  cbLength, cbScale, cbRadix, cbNullable;

/* retrieve information about all the qualifiers,
   owners, and all columns of the EMPLOYEE table             */
rc = SQLColumns(hstmt, "POLITE", SQL_NTS,
  "OOT_SCH", SQL_NTS, "EMPLOYEE", SQL_NTS, NULL, 0);
if (rc == SQL_SUCCESS) {
  /* Bind columns in result set to buffer */
  SQLBindCol(hstmt, 1, SQL_C_CHAR,
        szQualifier, STR_LEN, &cbQualifier);
  SQLBindCol(hstmt, 2, SQL_C_CHAR, szOwner, STR_LEN, &cbOwner);
  SQLBindCol(hstmt, 3, SQL_C_CHAR,
        szTableName, STR_LEN,&cbTableName);
  SQLBindCol(hstmt, 4, SQL_C_CHAR,
        szColName, STR_LEN, &cbColName);
  SQLBindCol(hstmt, 5, SQL_C_SSHORT, &DataType, 0, &cbDataType);
  SQLBindCol(hstmt, 6, SQL_C_CHAR,
        szTypeName, STR_LEN, &cbTypeName);
  SQLBindCol(hstmt, 7, SQL_C_SLONG, &Precision, 0, &cbPrecision);
  SQLBindCol(hstmt, 8, SQL_C_SLONG, &Length, 0, &cbLength);
  SQLBindCol(hstmt, 9, SQL_C_SSHORT, &Scale, 0, &cbScale);
  SQLBindCol(hstmt, 10, SQL_C_SSHORT, &Radix, 0, &cbRadix);
```

```
  SQLBindCol(hstmt, 11, SQL_C_SSHORT, &Nullable, 0, &cbNullable);
  SQLBindCol(hstmt, 12, SQL_C_CHAR, szRemarks, REM_LEN,
       &cbRemarks);

  while(TRUE) {
    rc = SQLFetch(hstmt);
    if (rc == SQL_ERROR || rc == SQL_SUCCESS_WITH_INFO) {
      show_error( );
    }
    if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO){
      /* process fetched data ... */
    } else
      break;
  }
}
```

## SQLConnect

***Description:*** Loads the OOT ODBC driver and establishes a connection to a data source.

***Syntax:***

```
RETCODE SQLConnect(HDBC hdbc, UCHAR *szDSN, SWORD cbDSN,
    UCHAR *szUID, SWORD cbUID, UCHAR *szAuthStr,
    SWORD cbAuthStr);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`szDSN`
a data source name.

`cbDSN`
the length of szDSN.

`szUID`
a user identifier.

`cbUID`
the length of szUID string.

`szAuthStr`
an authentication string.

`cbAuthStr`
the length of szAuthStr.

***Comments:*** If an existing Personal Oracle Lite database is found, the driver tries to open a

connection to the database; otherwise, a new database is created in the current working directory.

If the user ID string is not specified, “OOT_SCH” will be the default user ID. Other valid schema anmes may also be used.

***Examples:***

```
HENV    henv;/* environment handle */
HDBC    hdbc;/* connection handle */
HSTMT   hstmt;/* statement handle */
RETCODE rc;/* ODBC function return code */
rc = SQLAllocEnv(&henv); /* alloc environment handle */
if (rc == SQL_SUCCESS) {
    rc = SQLAllocConnect(henv, &hdbc);
    if (rc == SQL_SUCCESS) {
    /* connect to data source "OOT" with user id
       "DORIS" and password "OBJECT") */
        rc = SQLConnect(hdbc, "OOT", SQL_NTS, "DORIS",
            SQL_NTS, "OBJECT", SQL_NTS);
    if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO) {
      rc = SQLAllocStmt(hdbc, &hstmt);
      if (rc == SQL_SUCCESS) {
        /* do something with statement ... */
        SQLFreeStmt(hstmt, SQL_DROP);
      }
      SQLDisconnect(hdbc);
    }
    SQLFreeConnect(hdbc);
  }
  SQLFreeEnv(henv);
}
```

## SQLDescribeCol

***Description:*** Returns the result descriptor - column name, type, precision, scale, and nullability - for one column in the result set.

***Syntax:***

```
RETCODE SQLDescribeCol(HSTMT hstmt, UWORD icol, UCHAR *szColName,
    SWORD cbColNameMax, SWORD *pcbColName, SWORD *pfSqlType,
```

```
    UDWORD *pcbColDef, SWORD *pibScale, SWORD *pfNullable);
```

***Key words and parameters:***

`hstmt`

a statement handle.

`icol`

a column position in the result set, ordered sequentially from left to right, starting at 1.

`szColName`

a pointer to buffer for returning the column name. If the column is unnamed or the column name can not be determined, the driver returns an empty string.

`cbColNameMax`

the maximum length of the szColName buffer.

`pcbColName`

the total number of bytes (excluding the null termination byte) available for return in szColName. If this value is greater than or equal to cbColNameMax , the column name szColName is truncated to cbColNameMax-1 bytes.

`pfSqlType`

the ODBC SQL data type of the column. The OOT ODBC driver returns one of the following: SQL_BIGINT, SQL_CHAR, SQL_DATE, SQL_DECIMAL, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_LONGVARCHAR, SQL_REAL, SQL_SAMLLINT, SQL_TIME, SQL_TIMESTAMP, or SQL_VARCHAR.

`pcbColDef`

the precision of the column on the data source. If the precision can not be determined, the driver returns 0.

`pibScale`

the scale of the column in the result set. If the scale can not be determined or is not applicable, the driver returns 0.

`pfNullable`

an indicator that shows whether the column allows NULL values.

***Comments:***

An application typically calls `SQLDescribeCol` after a call to `SQLPrepare,` and possibly before or after an associated call to `SQLExecute`. An application can also call `SQLDescribeCol` after a call to `SQLExecDirect`, `SQLColumns,` or `SQLTables`. `SQLDescribeCol` returns the column name, type, and length generated by a `SELECT` statement. If the column is an expression, `szColName` contains a string name defined by the Personal Oracle Lite SQL server.

***Examples:***

```
#define NAME_LEN 30

HSTMT hstmt;/* statement handle */

UCHAR ColName[NAME_LEN];

SWORD ColNameLen;

SWORD ColType;

UDWORD  ColPrec;
```

```
SWORD ColScale;
SWORD Nullable;
RETCODE rc;   /* ODBC function return code */

rc = SQLPrepare(hstmt, "SELECT * FROM EMPLOYEE", SQL_NTS);
/* Describe the first column of the select statement */
rc = SQLDescribeCol(hstmt, 1, ColName, NAME_LEN,
  &ColNameLen,&ColType,&ColPrec,&ColScale,&Nullable);
```

## SQLDisconnect

***Description:*** Closes the connection associated with a specific connection handle.

***Syntax:***

```
RETCODE SQLDisconnect(HDBC hdbc);
```

***Key words and parameters:***

`hdbc`
a connection handle.

***Comments:*** If an application calls `SQLDisconnect` while there is an incomplete transaction associated with the connection handle, OOT ODBC driver returns `SQLSTATE 25000` (invalid transaction state). In this case, the state of the transaction is unchanged and the connection remains open. An incomplete transaction is one that has not been committed or rolled back with `SQLTransact`. If an application calls `SQLDisconnect` before it has freed all statement handles associated with the connection, the driver frees all these statements after it successfully disconnects from the data source.

***Examples:*** See `SQLConnect`.

## SQLDriverConnect

***Description:*** An alternative to `SQLConnect`. It is useful when the connect operation requires more information from the client application. `SQLDriverConnect` provides the following connection options:

- Establish a connection using a connection string that contains the data source name, one or more user IDs, one or more passwords, and other information required by the data source.
- Establish a connection using a partial connection string or no additional information; in this case, the driver manager and the driver can each prompt the user for connection information.

- Establish a connection to a data source that is not defined in the `ODBC.INI` file or registry. If the application supplies a partial connection string, the driver can prompt the user for connection information. Once a connection is established, `SQLDriverConnect` returns the completed connection string. The application can use this string for subsequent connection requests.

***Syntax:***

```
RETCODE SQLDriverConnect(HDBC hdbc, HWND hwnd,
    UCHAR *szConnStrIn, SWORD cbConnStrIn,
    UCHAR *szConnStrOut, SWORD cbConnStrOutMax,
    SWORD *pcbConnStrOut, UWORD fDriverCompletion);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`hwnd`
a window handle. The application can pass the handle of the parent window, if applicable, or it can pass a null pointer if either the window handle is not applicable or if SQLDriverConnect will not present any dialog boxes.

`szConnStrIn`
a full connection string, a partial connection string, or an empty string.

`cbConnStrIn`
the length of szConnStrIn.

`szConnStrOut`
a pointer to a buffer for the completed connection string. Upon successful connection to the target data source, this buffer contains the completed connection string. An application should allocate at least 255 bytes for this buffer.

`cbConnStrOutMax`
the maximum length of the szConnStrOut buffer.

`pcbConnStrOut`
a pointer to the total number of bytes (excluding the null termination byte) available for return in szConnStrOut. If the number of bytes available for return is greater than or equal to cbConnStrOutMax, the completed connection string in szConnStrOut is truncated to cbConnStrOutMax-1 bytes.

`fDriverCompletion`
a flag that indicates whether the driver manager or driver must prompt for more connection information: SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE, SQL_DRIVER_COMPLETE_REQUIRED, and SQL_DRIVER_NOPROMPT.

***Comments:*** In the current release, the OOT ODBC driver allows an application to make a connection to a database residing outside of any installed data source.

***Examples:*** See `SQLGetConnectOption`.

## SQLError

***Description:*** Returns error or status information.

***Syntax:***

```
RETCODE SQLError(HENV henv, HDBC hdbc, HSTMT hstmt,
    UCHAR *szSqlState, SDWORD *pfNativeError,
    UCHAR *szErrorMsg,SWORD cbErrorMsgMax,
    SWORD *pcbErrorMsg);
```

***Key words and parameters:***

`henv`
an environment handle or SQL_NULL_HENV.

`hdbc`
a connection handle or SQL_NULL_HDBC.

`hstmt`
a statement handle or SQL_NULL_HSTMT.

`szSqlState`
SQLSTATE as a null terminated string.

`pfNativeError`
an error code specific to OOT ODBC driver.

`szErrorMsg`
a pointer to a buffer for error message text.

`cbErrorMsgMax`
the maximum length of the szErrorMsg buffer.

`pcbErrorMsg`
a pointer to the total number of bytes (excluding the null termination byte) available for return in szErrorMsg. If the number of bytes available for return is greater than or equal to cbErrorMsgMax, the error message text in szErrorMsg is truncated to cbErrorMsgMax-1 bytes.

***Comments:***

An application typically calls `SQLError` when a previous call to an ODBC function returns `SQL_ERROR` or `SQL_SUCCESS_WITH_INFO`. When `SQLError` is called, the driver finds the rightmost non-null handle argument, and decides where to find the error information to return.

To access error information associated with the statement handle, the application must make sure that the third parameter `hstmt` is not null. To access error information associated with the connection handle, the application must pass in a valid connection handle as the second parameter and a null statement handle as the third parameter. Similarly, to access error information associated with the environment handle, an application must pass in a valid environment handle as the first parameter, and null handles as the second and third parameters.

To retrieve multiple errors resulting from a function call, an application must make multiple calls to `SQLError`. For each error, the driver returns `SQL_SUCCESS` and removes that error from the list of available errors.

When there is no additional information for the rightmost non-null handle, `SQLError` returns `SQL_NO_DATA_FOUND`. In this case, `szSqlState` equals 00000 (Success),

pfNativeError is undefined, cbErrorMsg equals 0, and szErrorMsg contains a single null termination byte (unless cbErrorMsgMax equals 0).

An error is removed from the structure associated with a handle when SQLError is called with that handle and the error is returned. All errors maintained for a given handle are removed when that handle is used in a subsequent function call.

***Examples:*** See SQLBindCol.

## SQLExecDirect

***Description:*** Executes a preparable statement, using the current values of the parameter marker variables if any parameters exist in the statement.

***Syntax:***

```
RETCODE SQLExecDirect(HSTMT hstmt, UCHAR *szSqlStr, SDWORD cbSqlStr);
```

***Key words and parameters:***

hstmt
  a statement handle.
szSqlStr
  a SQL statement to be executed.
cbSqlStr
  the length of szSqlStr.

***Comments:*** An application calls SQLExecDirect to issue a SQL statement to the Personal Oracle Lite SQL server for execution. The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark '?' into the SQL statement at the appropriate position.

If SQLExecDirect encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The application can then supply the data by using SQLParamData to obtain information about the data needed and by passing in the actual data with SQLPutData. The final call to SQLParamData (that is, the call after all the necessary data have been passed in) completes the execution.

***Examples:*** See SQLBindCol.

## SQLExecute

***Description:*** Executes a prepared statement, using the current values of the parameter marker variables if there exist any parameter markers in the statement.

***Syntax:***

```
RETCODE SQLExecute(HSTMT hstmt);
```

***Key words and parameters:***

`hstmt`
a statement handle.

***Comments:*** `SQLExecute` executes a statement prepared by `SQLPrepare`. After the application processes or discards the results from a call to `SQLExecute`, the application can call `SQLExecute` again with new parameter values.

To execute a `SELECT` statement more than once, the application must call `SQLFreeStmt` with the `SQL_CLOSE` parameter before re-issuing the `SELECT` statement.

If `SQLExecDirect` encounters a data-at-execution parameter, it returns `SQL_NEED_DATA`. The application can then supply the data by using `SQLParamData` to obtain information about the data needed and by passing in the actual data with `SQLPutData`. The final call to `SQLParamData` (that is, the call after all the necessary data have been passed in) completes the execution.

***Examples:*** See `SQLBindParameter`.

## SQLExtendedFetch

***Description:*** Returns one or more rows, in the form of an array, for each bound column.

```
Syntax:

RETCODE SQLExtendedFetch(HSTMT hstmt, UWORD fFetchType,

  SDWORD irow, UDWORD FAR *pcrow, UDWORD FAR *rgfRowStatus);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`fFetchType`
the type of fetch (that is, how to scroll through the result set). Valid values are: SQL_FETCH_NEXT, SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_PRIOR, SQL_FETCH_ABSOLUTE, SQL_FETCH_RELATIVE. If the value of the SQL_CURSOR_TYPE statement option is SQL_CURSOR_FORWARD_ONLY, the fFetchType argument must be SQL_FETCH_NEXT.

`irow`
the number of rows to fetch. For the SQL_FETCH_ABSOLUTE fetch type, SQLExtendedFetch returns the rowset starting at the row number given by the irow argument. For the SQL_FETCH_RELATIVE fetch type, SQLExtendedFetch returns the rowset starting irow rows from the first row in the current rowset. The irow argument is ignored for the SQL_FETCH_NEXT, SQL_FETCH_PRIOR, SQL_FETCH_FIRST, and SQL_FETCH_LAST fetch types.

`pcrow`
the number of rows actually fetched.

`rgfRowStatus`
an array of status values. OOT ODBC driver cannot detect changes to data. Therefore, this argument is not updated to reflect the change in the row status.

***Comments:*** SQLExtendedFetch returns a rowset of data to the caller. A caller specifies the number of rows in the rowset by calling SQLSetStmtOption with the SQL_ROWSET_SIZE statement option.

If any columns in the result set have been bound with SQLBindCol, the driver converts the data for the bound columns as necessary and stores it in the locations bound to those columns. In the current release, only Column-Wise binding is supported.

A caller requests Column-Wise binding by setting the SQL_BIND_TYPE statement option to SQL_BIND_BY_COLUMN. For each bound column in the result set, the caller:

3. Allocates an array of data buffers to store the rows in the rowset. The size of each buffer is the maximum size of the C data that can be returned for the column.
4. Allocates an array of SDWORDs to hold the number of bytes available to return for each row in the column.
5. Calls SQLBindCol: The rgbValue argument specifies the address of the buffer. The cbValueMax argument specifies the size of each buffer in the data buffer array. The pcbValue argument specifies the address of the array that stores the number of bytes returned.
6. Calls SQLExtendedFetch.

***Examples:***

```
#define ROWS 150
#define NAME_LEN 30
#define BDAY_LEN 11

UCHAR szName[ROWS][NAME_LEN];
UCHAR szBirthday[ROWS][BDAY_LEN];
SWORD sAge[ROWS];
SDWORD  cbName[ROWS], cbAge[ROWS], cbBirthday[ROWS];
UDWORD  crow, irow;
UWORD rgfRowStatus[ROWS];
RETCODE rc;   /* ODBC function return code */

SQLSetStmtOption(hstmt, SQL_CURSOR_TYPE, SQL_CURSOR_DYNAMIC);
SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWS);
```

```
rc = SQLExecDirect(hstmt, "SELECT NAME, AGE,
  BIRTHDAY FROM EMPLOYEE ORDER BY  1", SQL_NTS);

if (rc == SQL_SUCCESS) {
  SQLBindCol(hstmt, 1, SQL_C_CHAR,
         szName, NAME_LEN, cbName);
  SQLBindCol(hstmt, 2, SQL_C_SSHORT, sAge, 0, cbAge);
  SQLBindCol(hstmt, 3, SQL_C_CHAR,
         szBirthday, BDAY_LEN, cbBirthday);

  /* Fetch the rowset data  */
  while (TRUE) {
    rc = SQLExtendedFetch(hstmt,
        SQL_FETCH_NEXT, 1, &crow, rgfRowStatus);
    if (rc != SQL_SUCCESS) { /* error handling */ }
      for (irow = 0; irow < crow; irow++) {
      /* process the row ... */
    }
  }
}
```

## SQLFetch

***Description:*** Fetches a row of data from a result set.

***Syntax:***

```
RETCODE SQLFetch(HSTMT hstmt);
```

***Key words and parameters:***

`hstmt`
a statement handle.

***Comments:*** `SQLFetch` advances the current cursor of a statement to the next qualified row. If the application has called `SQLBindCol` to bind columns, `SQLFetch` places data into the buffer locations specified by previous calls to `SQLBindCol`. `SQLFetch` accesses column data in the left-to-right order. If the application has not called `SQLBindCol` to bind any column, `SQLFetch` simply moves the cursor to the next row.

After each fetch, a `pcbValue` specified in `SQLBindCol` contains the number of bytes available to return for the column. If `SQL_MAX_LENGTH` has been specified with `SQLSetStmtOption` and the number of bytes available to return is greater

than SQL_MAX_LENGTH, the pbValue is set to SQL_MAX_LENGTH.

If rgbvalues is not large enough to hold the entire result, the driver places part of the value and returns SQL_SUCCESS_WITH_INFO. If the data value for a column is NULL, the driver sets pcbValue to SQL_NULL_DATA. When the result set is exhausted, the driver returns SQL_NO_DATA_FOUND.

After the execution of a SELECT statement, a cursor is opened internally by the driver. The cursor handle is maintained in the handle for the SELECT statement by the driver.

***Examples:*** See SQLBindCol and SQLColumns.

## SQLForeignKeys

***Description:*** Returns as result sets a list of foreign keys in the table that refer to primary keys in other tables and a list of foreign keys in other tables that refer to the primary key in the specified table.

***Syntax:***

```
RETCODE SQLPrimaryKeys (HSTMT hstmt,
  UCHAR FAR *szPkTableQualifier, SWORD cbPkTableQualifier,
  UCHAR FAR *szPkTableOwner, SWORD cbPkTableOwner,
  UCHAR FAR *szPkTableName, SWORD cbPkTableName,
  UCHAR FAR *szFkTableQualifier, SWORD cbFkTableQualifier,
  UCHAR FAR *szFkTableOwner, SWORD cbFkTableOwner,
  UCHAR FAR *szFkTableName, SWORD cbFkTableName);
```

***Key words and parameters:***

hstmt
  a statement handle.
szPkTableQualifier
  a qualifier name.
cbPkTableQualifier
  the length of szPkTableQualifier.
szPkTableOwner
  name of table owner.
cbPkTableOwner
  the length of szPkTableOwner.
szPkTableName
  a string search pattern for table names.
cbPkTableName
  the length of szPkTableName.
szFkTableQualifier
  a qualifier name.

`cbFkTableQualifier`
the length of szFkTableQualifier.
`szFkTableOwner`
name of table owner.
`cbFkTableOwner`
the length of szFkTableOwner.
`szFkTableName`
a string search pattern for table names.
`cbFkTableName`
the length of szFkTableName.

***Comments:*** If `szPkTableName` contains a table name, `SQLForeignKeys` returns a result set containing the primary keys of the specified table and all of the foreign keys that refer to it. The result set is ordered by:

- `FKTABLE_QUALIFIER`
- `FKTABLE_OWNER`
- `FKTABLE_NAME`
- `KEY_SEQ`

If `szFkTableName` contains a table name, `SQLForeignKeys` returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer. The result set is ordered by `PKTABLE_QUALIFIER`, `PKTABLE_OWNER`, `PKTABLE_NAME`, and `KEY_SEQ`.

If both `szPkTableName` and `szFkTableName` contain table names, `SQLForeignKeys` returns the foreign keys in the table specified in `szFkTableName` that refer to the primary key of the table specified in `szPkTableName`. There must be one key at most.

The following table lists the columns in the result set:

| Column Name | Data Type | Comments |
|---|---|---|
| PKTABLE_QUALIFIER | Varchar(128) | primary key table qualifier identifier. |
| PKTABLE_OWNER | Varchar(128) | primary key table owner identifier. |
| PKTABLE_NAME | Varchar(128) not NULL | primary key table identifier. |
| PKCOLUMN_NAME | Varchar(128) not NULL | primary key column identifier. |
| FKTABLE_QUALIFIER | Varchar(128) | foreign key table qualifier identifier . |
| FKTABLE_OWNER | Varchar(128) | foreign key table owner identifier. |
| FKTABLE_NAME | Varchar(128) not NULL | foreign key table identifier. |
| FKCOLUMN_NAME | Varchar(128) not NULL | foreign key column identifier. |
| KEY_SEQ | Smallint not NULL | column sequence number in key (starting with 1). |
| UPDATE_RULE | Smallint | action to be applied to the foreign key when the SQL operation is UPDATE:SQL_CASCADE, SQL_RESTRICT, SQL_SET_NULL; NULL if not applicable to the data source. |
| DELETE_RULE | Smallint | action to be applied to the foreign key when the SQL operation is DELETE:SQL_CASCADE, SQL_RESTRICT, SQL_SET_NULL; NULL if not applicable to the data source. |
| PK_NAME | Varchar(128) | primary key identifier. |
| FK_NAME | Varchar(128) | foreign key identifier. |

***Examples:*** See SQLPrimaryKeys.

## SQLFreeConnect

***Description:*** Releases a connection handle and frees all memory space associated with it.

***Syntax:***
```
RETCODE SQLFreeConnect(HDBC hdbc);
```

***Key words and parameters:***

`hdbc`
a connection handle.

***Comments:*** Before calling `SQLFreeConnect`, an application must call `SQLDisconnect` with the connection handle. Otherwise, `SQLFreeConnect` returns `SQL_ERROR` and the connection handle remains valid. As a side effect, `SQLDisconnect` automatically drops all statement handles open on the connection.

***Examples:*** See `SQLConnect`.

## SQLFreeEnv

***Description:*** Frees an environment handle and releases all its associated memory.

***Syntax:***
```
RETCODE SQLFreeEnv(HENV henv);
```

***Key words and parameters:***

`henv`
an environment handle.

***Comments:*** Before calling `SQLFreeEnv`, an application must call `SQLFreeConnect` for all connection handles allocated under the environment handle. Otherwise, `SQLFreeEnv` returns `SQL_ERROR.` In this case, the environment handle and all its active connection handles remain valid.

***Examples:*** See `SQLConnect`.

## SQLFreeStmt

***Description:*** Stops processing associated with a statement handle, closes open cursors associated with the statement handle, discards pending results, and optionally frees all resources associated with the statement handle.

***Syntax:***
```
RETCODE SQLFreeStmt(HSTMT hstmt, UWORD fOption);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`fOption`
one of the following options:

1. SQL_CLOSE to close the cursor associated with the statement handle and to discard all pending results
2. SQL_DROP to release the statement handle, free all the resources associated with it, close the cursor, and discard all pending results
3. SQL_UNBIND to release column buffers bound to the statement handle
4. SQL_RESET_PARAMS to release all parameter buffers for the statement handle

***Comments:*** An application can call `SQLFreeStmt` to terminate the processing of a `SELECT` statement, with or without invalidating the statement handle.

***Examples:*** See `SQLConnect`.

## SQLGetConnectOption

***Description:*** Returns the current setting of a connection option.

***Syntax:***
```
RETCODE SQLGetConnectOption(HDBC hdbc, UWORD fOption, PTR pvParam);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`fOption`
an option to retrieve.

`pvParam`
a value associated with fOption. Depending on fOption, a 32-bit integer value or a pointer to a null-terminated character string is returned.

***Comments:*** Refer to the `SQLSetConnectOption` for a list of supported options.

***Examples:***
```
HWND  hwnd;/* window handle */
HDBC  henv;/* environment handle */
HDBC  hdbc;/* connection handle */
HSTMT hstmt;/* statement handle */
UDWORD  cbOption;/* connection option */
RETCODE rc;   /* ODBC function return code */

/* ... set up henv and hwnd ... */
SQLAllocConnect(henv, &hdbc);
rc = SQLDriverConnect(hdbc, hwnd,
    "DataDirectory=D:\\OOT\\DB", SQL_NTS, szConnStrOut,
    CONNECT_STRING_LEN, SQL_DRIVER_NOPROMPT);
```

```
if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO) {
  rc = SQLAllocStmt(hdbc,&hstmt);
  if (rc == SQL_SUCCESS) {
    SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, (PTR)&cbOption);
    if (cbOption != SQL_AUTOCOMMIT_ON) {
      SQLTransact(SQL_NULL_HENV, hdbc, SQL_COMMIT);
    }
    SQLFreeStmt(hstmt, SQL_DROP);
  }
  SQLDisconnect(hdbc);
}
SQLFreeConnect(hdbc);
```

## SQLGetCursorName

***Description:*** Returns the cursor name associated with a statement handle.

***Syntax:***

```
RETCODE SQLSetCursorName(HSTMT hstmt, UCHAR FAR * szCursor,
  SWORD cbCursorMax, SWARD FAR *pcbCursor);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`szCursor`
a pointer to storage for the cursor name.

`cbCursorMax`
the length of szCursor.

`pcbCursor`
total number of bytes (excluding the null termination byte) returned in szCursor.

***Comments:*** The maximum length of a cursor name is 18 characters. SQLGetCursorName returns the cursor name that has been set either explicitly or implicitly.

***Examples:***

```
HSTMT hstmt;/* statement handle */
UCHAR Name[MAX_NAME_LEN];/* for returning cursor name */
SDWORD  NumBytes = 0;/* size of cursor name */
RETCODE rc;     /* ODBC function return code */

rc = SQLSetCursorName(hstmt, "EMP_CUR", SQL_NTS);
if(rc != SQL_SUCCESS) { /* error handling */ }
```

```
rc = SQLGetCursorName(hstmt, Name, MAX_NAME_LEN, &NumBytes);
if(rc != SQL_SUCCESS) { /* error handling */ }
```

## SQLGetData

***Description:*** Returns result data for a single unbound column in the current row. The application must call `SQLFetch` before it calls this function. It is possible to blend uses of `SQLBindCol` for some columns and uses of `SQLGetData` for others within the same row. This function can be used to retrieve data values incrementally from `SQL_LONGVARCHAR` columns.

***Syntax:***

```
RETCODE SQLGetData(HSTMT hstmt, UWORD icol, SWORD fCType,
    PTR rgbValue, SWORD cbValueMax, SWORD *pcbValue);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`icol`
a column position in the result data, ordered sequentially left to right, starting at 1.

`fCType`
the C data type of the result data. OOT ODBC driver supports the following types: SQL_C_CHAR, SQL_C_DATE, SQL_C_DEFAULT, SQL_C_DOUBLE, SQL_C_FLOAT, SQL_C_SLONG, SQL_C_SSHORT, SQL_C_TIME, SQL_C_TIMESTAMP, and SQL_C_DEFAULT. SQL_C_DEFAULT specifies that data be converted to its default C data type.

`rgbValue`
a pointer to a buffer for receiving the data.

`cbValueMax`
the maximum length of the rgbValue buffer. For character data, rgbValue must also include space for the null-termination byte. For character and binary C data, cbValueMax determines the amount of data that can be received in a single call to SQLGetData. For all other types of C data, cbValueMax is ignored; the driver assumes that the size of rgbValue is the size of the C data type specified by fCType and returns the entire data value.

`pcbValue`
SQL_NULL_DATA, the total number of bytes (excluding the null termination byte for character data) available to return in rgbValue prior to the current call to SQLGetData, or SQL_NO_TOTAL if the number of available bytes can not be determined. For character data, if pcbValue is SQL_NO_TOTAL or is greater than or equal to cbValueMax, the data in rgbValue is truncated to cbValueMax-1 bytes and is null-terminated by the driver. For all other data types, the value of cbValueMax is ignored and the driver assumes the size of rgbValue is the size of the C data type specified by fCType.

***Comments:*** The function sets `pcbValue` to the number of bytes that were available in the result column prior to the current call to `SQLGetData`. If the data value for the column is `NULL`, the driver set `pcbValue` to `SQL_NULL_DATA`.

When an application calls this function to retrieve data from a LONG VARCHAR column, the driver returns SQL_SUCCESS_WITH_INFO. A subsequent call to SQLError returns SQLSTATE 01004 (data truncated). The application can then specify the same column position to retrieve subsequent parts of the data until SQLGetData returns SQL_SUCCESS, indicating that all data for the column has been retrieved.

SQLGetData returns SQL_NO_DATA_FOUND when it is called for a column after all of the data has been retrieved and before data is retrieved for a subsequent column. The application can ignore excess data by proceeding to the next result column.

***Examples:***

```
#define NAME_LEN 30
HDBC  hdbc;   /* connection handle */
HSTMT hstmt;  /* statement handle */
SDWORDID;      /* employee ID */
char  szFirstName[NAME_LEN];/* first name */
char  szLastName[NAME_LEN];/* last name */

/* set up hdbc ... */
SQLAllocStmt(hdbc, &hstmt);
SQLExecDirect(hstmt,
  "SELECT ID, FIRST_NAME, LAST_NAME FROM EMPLOYEE", SQL_NTS);

/* for each employee row, get ID, first & last names */
while (SQLFetch(hstmt) != SQL_NO_DATA_FOUND) {
  SQLGetData(hstmt, 1, SQL_C_LONG, &cbID, 0, NULL);
  SQLGetData(hstmt, 2, SQL_C_CHAR, szFirstName, NAME_LEN, NULL);
  SQL(hstmt, 3, SQL_C_CHAR, szLastName, NAME_LEN, NULL);
  /* do something with the data ... */
}
```

## SQLGetInfo

***Description:*** Returns general information about the OOT ODBC driver and the data source associated with a connection handle.

***Syntax:***

```
RETCODE SQLGetInfo(HDBC hdbc, UWORD fInfoType, PTR rgbInfoValue,
    SWORD cbInfoValueMax, SWORD *pcbInfoValue);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`fInfoType`
the type of information of interest.

`rgbInfoValue`
a pointer to a buffer for receiving the information. Depending on the fInfoType requested, the information returned is one of the following:

- null-terminated character string
- 16-bit integer value
- 32-bit flag
- 32-bit binary value

`cbInfoValueMax`
the maximum length of the rgbInfoValue buffer.

`pcbInfoValue`
the total number of bytes (excluding the null termination byte for character data) available to return in rgbInfoValue. For character data, if the number of bytes available to return is greater than or equal to cbInfoValueMax, the data in rgbInfoValue is truncated to cbInfoValueMax-1 bytes and is null-terminated by the driver. For all other types of data, the value of cbValueMax is ignored and the driver assumes the size of rgbValue is 32 bits.

***Comments:***

The format of the information returned in `rgbInfoValue` depends on the `fInfoType` requested. `SQLGetInfo` returns information in one of five different formats:

- null-terminated character string
- 16-bit integer value
- 32-bit bitmask
- 32-bit integer value
- 32-bit binary value

The application must cast the value returned in `rgbInfoValue` according to the information type. For a detailed description of each information type, please refer to the function reference for `SQLGetInfo` in *ODBC 2.0 Programmer's Reference and SDK Guide*.

***Examples:***

```
#define DSN_LEN30

HDBC  hdbc; /* statement handle */

char  szDSName[DSN_LEN];/* name of data source */

SDWORD  cbDSName;/* size of data source name */
```

```
SQLGetInfo(hdbc, SQL_DATA_SOURCE_NAME,
    (PTR)szDSName, DSN_LEN, &cbDSName);
/* do something with data source name in szDSName ...*/
```

## SQLGetStmtOption

***Description:*** Returns the current value of a statement option.

***Syntax:***
```
RETCODE SQLGetStmtOption(HSTMT hstmt, UWORD fOption, PTR pvParam);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`fOption`
an option to retrieve.

`pvParam`
a pointer to a buffer for returning the option value.

***Comments:*** See `SQLSetStmtOption` for a list of available options.

***Examples:***
```
HSTMT hstmt;/* statement handle */
UDWORD  Concurrency;/* for returning option */
RETCODE rc;     /* ODBC function return code */

rc = SQLSetStmtOption(hstmt, SQL_CONCURRENCY, SQL_CONCUR_ROWVER);
if(rc == SQL_SUCCESS_WITH_INFO) {
  rc = SQLGetStmtOption(hstmt, SQL_CONCURRENCY, &Concurrency);
  if(Concurrency != SQL_CONCUR_ROWVER) {
    /* handle changed option value */
  }
}
```

## SQLGetTypeInfo

***Description:*** Returns information about data types supported by the Personal Oracle Lite SQL server in the form of a standard result set.

***Syntax:***
```
RETCODE SQLGetTypeInfo(HSTMT hstmt, SWORD fSqlType);
```

***Key words and parameters:***

`hstmt`
a statement handle for the result set.

`fSqlType`

the SQL data type. This must be one of the following values: SQL_BIGINT, SQL_BINARY, SQL_BIT, SQL_CHAR, SQL_DATE, SQL_DECIMAL, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_LONGVARBINARY, SQL_LONGVARCHAR, SQL_NUMERIC, SQL_REAL, SQL_SMALLINT, SQL_TIME, SQL_TIMESTAMP, SQL_TINYINT, SQL_VARBINARY, SQL_VARCHAR, or SQL_ALL_TYPES. In case SQL_ALL_TYPES is specified, this function returns information about all the supported types.

***Comments:*** `SQLGetTypeInfo` returns the results as a standard result set, ordered by `DATA_TYPE` and `TYPE_NAME`. The following table lists the columns in the result set.

| Column Name | Data Type | Comments |
|---|---|---|
| TYPE_NAME | Varchar(128) not NULL | Data source-dependent data type name, such as, "CHAR", "VARCHAR", "MONEY", "LONG VARBINARY", or "CHAR ( ) FOR BIT DATA". An application must use this name in CREATE TABLE and ALTER TABLE statements. |
| DATA_TYPE | Smallint not NULL | SQL data type. This can be an ODBC SQL data type or a driver-specific SQL data type. For a list of valid ODBC SQL data types, see SQL Data Types. |
| PRECISION | Integer | The maximum precision of the data type on the data source. NULL is returned for data types where precision is not applicable. |
| LITERAL_PREFIX | Varchar(128) | Character or characters used to prefix a literal. For example, a single quote ( ’ ) is used for character data types, and 0x is used for binary data types. NULL is returned for data types where a literal prefix is not applicable. |
| LITERAL_SUFFIX | Varchar(128) | Character or characters used to terminate a literal. For example, a single quote ( ’ ) is used for character data types. NULL is returned for data types where a literal suffix is not applicable. |
| CREATE_PARAMS | Varchar(128) | Parameters for a data type definition. For DECIMAL, "precision,scale" is returned. For VARCHAR, "max length" is returned. For data types with no parameters, such as INTEGER, NULL is returned. |
| NULLABLE | Smallint not NULL | States whether the data type accepts a NULL value: SQL_NO_NULLS if the data type does not accept NULL values, and SQL_NULLABLE if the data type accepts NULL values. SQL_NULLABLE_UNKNOWN if it is not known if the column accepts NULL values. |
| CASE_SENSITIVE | Smallint not NULL | States whether a character data type is case sensitive in collations and comparisons: TRUE if the data type is a character data type and is case sensitive; FALSE if the data type is not a character data type or is not case sensitive. |
| SEARCHABLE | Smallint not NULL | States how the data type is used in a WHERE clause: SQL_UNSEARCHABLE if the data type cannot be used in a WHERE clause, SQL_LIKE_ONLY if the data type can be used in a WHERE clause only with the LIKE predicate, SQL_ALL_EXCEPT_LIKE if the data type can be used in a WHERE clause with all comparison operators except LIKE, and SQL_SEARCHABLE if the data type can be used in a WHERE clause with any comparison operator. |
| UNSIGNED_ ATTRIBUTE | Smallint | States whether the data type is unsigned: TRUE if the data type is unsigned, FALSE if the data type is signed, and NULL if the attribute is not applicable to the data type or the data type is not numeric. |
| MONEY | Smallint not NULL | States whether the data type is a money data type: TRUE if it is a money data type; FALSE if it is not. |

| | | |
|---|---|---|
| AUTO_ INCREMENT | Smallint | States whether the data type is autoincrementing: TRUE if the data type is autoincrementing; FALSE if the data type is not autoincrementing. NULL is returned if the attribute is not applicable to the data type or the data type is not numeric. An application can insert values into a column having this attribute, but can not update the values in the column. |
| LOCAL_TYPE_ NAME | Varchar(128) | Localized version of the data source-dependent name of the data type. NULL is returned if a localized name is not supported by the data source. This name is intended for display only, such as in dialog boxes. |
| MINIMUM_SCALE | Smallint | The minimum scale of the data type on the data source. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain this value. For example, an SQL_TIMESTAMP column might have a fixed scale for fractional seconds. NULL is returned where the scale is not applicable. For more information, see Precision, Scale, Length, and Display Size. |
| MAXIMUM_SCALE | Smallint | The maximum scale of the data type on the data source. NULL is returned where the scale is not applicable. If the maximum scale is not defined separately on the data source, but is instead defined to be the same as the maximum precision, this column contains the same value as the PRECISION column. |

***Examples:***

```
HSTMT hstmt;    /* statement handle */
UDWORD  cbPrecision;/* precision of column */

/* retrieve precision of a LONG VARCHAR type */
SQLGetTypeInfo(hstmt, SQL_LONGVARCHAR);
SQLBindCol(hstmt,3, SQL_C_LONG, &cbPrecision, 4, NULL);
SQLfetch(hstmt);
/* do something with precision in cbPrecision ... */
```

## SQLNativeSQL

***Description:*** Returns a translated SQL string.

***Syntax:***

```
RETCODE SQLNativeSql(HDBC hdbc, UCHAR FAR *szSqlStrIn,
  SDWORD cbSqlStrIn, UCHAR FAR *szSqlStr,
  SDWORD cbSqlStrMax, SWORD FAR *pcbSqlStr);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`szSqlStrIn`
a SQL text to be translated.

`cbSqlStrIn`
the length of the szSqlStrIn text string.

`szSqlStr`
a pointer to storage for the translated SQL string.

`cbSqlStrMax`

the maximum length of the szSqlStr buffer.

`pcbSqlStr`
the total number of bytes (excluding the null termination byte) available to return in szSqlStr.

***Comments:*** When SQLNativeSql returns SQL_SUCCESS, the output translated string is identical to the input SQL text string. SQLNativeSql checks if the given SQL statement is preparable.

***Examples:***

```
HDBC  hdbc; /* connection handle */
SDWORD  NumRetBytes = 0;/* bytes in translated string */
UCHAR NativeText[MAX_STMT_LEN];
RETCODE rc;   /* ODBC function return code */


rc = SQLNativeSql(hdbc,
    "INSERT INTO EMPLOYEE VALUES(DORIS, 30)", SQL_NTS,
    NativeText, MAX_STMT_LEN, &NumRetBytes);
```

## SQLNumResultCols

***Description:*** Returns the number of columns in a result set.

***Syntax:***

```
RETCODE SQLNumResultCols(HSTMT hstmt, SWORD *pcol);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`pcol`
the number of columns in the result set.

***Comments:*** `SQLNumResultCols` can be called successfully only when the statement handle is in the prepared, executed state. If the prepared SQL statement associated with the handle does not return a result set, `SQLNumResultCols` returns 0.

***Examples:***

```
HSTMT hstmt;/* statement handle */
SWORD cbColCount;/* # of columns */


SQLNumResultCols(hstmt, &cbColCount);
```

## SQLParamData

***Description:*** To be used in conjunction with `SQLPutData` to supply parameter data at statement execution time.

***Syntax:***

```
RETCODE SQLParamData(HSTMT hstmt, PTR *prgbValue);
```

***Key words and parameters:***

`hstmt`

a statement handle.

`prgbValue`

a pointer to a buffer for the value specified for the rgbValue argument in SQLBindParameter (for parameter data) or the address of the rgbValue buffer specified in SQLBindCol (for column data).

***Comments:***

See `SQLBindParameter`.

***Examples:***

```
#defineBUF_LEN128

HSTMT hstmt;/* statement handle */

UCHAR szComment[BUF_LEN]; /* buffer for COMM */

SDWORD  cbComment;/* length of COMM */

RETCODE rc;     /* ODBC function return code */


SQLExecDirect(hstmt, "CREATE TABLE EMPLOYEE (\
  ID INTEGER, FIRST_NAME CHAR(30), LAST_NAME \
  CHAR(30), COMM LONG VARCHAR)", SQL_NTS);
SQLPrepare(hstmt,
  "INSERT INTO EMPLOYEE VALUES (?, ?, ?, ?)", SQL_NTS);
SQLBindParameter(hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR,
  SQL_LONGVARCHAR, 0, 0, NULL, 0, NULL);


rc = SQLExecute(hstmt);
while (rc == SQL_NEED_DATA) {
  rc = SQLParamData(hstmt, NULL);
  if (rc == SQL_NEED_DATA) {
    /* repeatedly put each portion of comments */
    GetMoreComments(szComment);
    while (cbComment = strlen(szComment)) {
      SQLPutData(hstmt, szComment, cbComment);
      GetMoreComments(szComment);
    }
  }
}
```

## SQLPrepare

***Description:*** Prepares a SQL string as a statement for future execution.

***Syntax:***

```
RETCODE SQLPrepare(HSTMT hstmt, UCHAR *szSqlStr, SWORD cbSqlStr);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`szSqlStr`
a SQL text string.

`cbSqlStr`
the length of the SQL text string.

***Comments:*** An application calls `SQLPrepare` to send an SQL statement to the Personal Oracle Lite SQL server for preparation. The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark '?' into the SQL string at the appropriate position. Once a statement is prepared, the application uses its statement handle to refer to the statement in later function calls. The prepared statement associated with the statement handle can be repeatedly executed by calling `SQLExecute` until the statement handle is released with a call to `SQLFreeStmt` with the `SQL_DROP` option or until the statement handle is used in a subsequent call to `SQLPrepare`, `SQLExecDirect`, or one of the catalog functions (`SQLColumns`, `SQLTables`, and so on).

After the application prepares a statement, it can request information about the format of the result set. For maximum interoperability, an application must unbind all parameters that applied to an old SQL statement before preparing a new SQL statement on the same statement handle. This prevents errors that are due to old parameter information being applied to the new statement.

***Examples:***

```
#define NAME_LEN30
UCHAR szName[NAME_LEN];
SWORD sAge;
SDWORD  cbName = SQL_NTS, cbBirthday = 0;
SDWORD  cbAge = 0;
DATE_STRUCTdsBirthday;
RETCODE rc; /* ODBC function return code */


rc = SQLPrepare(hstmt, "INSERT INTO EMPLOYEE (NAME, \
    AGE, BIRTHDAY) VALUES (?, ?, ?)", SQL_NTS);
if (rc == SQL_SUCCESS) {
```

```
  /* Specify data types & buffers for Name, Age, Birthday */
  SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
    SQL_C_CHAR, SQL_CHAR, NAME_LEN, 0, szName, 0, &cbName);
  SQLBindParameter(hstmt, 2,SQL_PARAM_INPUT,
    SQL_C_SSHORT, SQL_SMALLINT, 0, 0, &sAge, 0, &cbAge);
  SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_DATE,
    SQL_DATE, 0, 0, &dsBirthday, 0, &cbBirthday);

  /* set up attribute values of the first row */
  strcpy(szName, "Smith, John D.");
  sAge = 40;
  dsBirthday.year = 1952;
  dsBirthday.month = 2;
  dsBirthday.day = 29;
  retcode = SQLExecute(hstmt); /* insert first row */

  /* set up attribute values of the second row */
  strcpy(szName, "Jones, Bob K.");
  sAge = 52;
  dsBirthday.year = 1940;
  dsBirthday.month = 3;
  dsBirthday.day = 31;
  rc = SQLExecute(hstmt); /* insert second row */
}
```

## SQLPrimaryKeys

***Description:*** Returns as a result set the names of columns that form a primary key for a table.

***Syntax:***

```
RETCODE SQLPrimaryKeys (HSTMT hstmt,
    UCHAR FAR *szTableQualifier, SWORD cbTableQualifier,
    UCHAR FAR *szTableOwner, SWORD cbTableOwner,
    UCHAR FAR *szTableName, SWORD cbTableName);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`szTableQualifier`
  a qualifier name.
`cbTableQualifier`
  the length of szTableQualifier.
`szTableOwner`
  name of table owner.
`cbTableOwner`
  the length of szTableOwner.
`szTableName`
  a string search pattern for table names.
`cbTableName`
  the length of szTableName.

***Comments:*** `SQLPrimaryKeys` returns the primary key information as a result set, ordered by `TABLE_QUALIFIER`, `TABLE_OWNER`, `TABLE_NAME`, and `KEY_SEQ`.

The following table lists the columns in the result set:

| Column Name | Data Type | Comments |
|---|---|---|
| TABLE_QUALIFIER | Varchar(128) | table qualifier identifier . |
| TABLE_OWNER | Varchar(128) | table owner identifier. |
| TABLE_NAME | Varchar(128) not NULL | table identifier. |
| COLUMN_NAME | Varchar(128) not NULL | key column identifier. |
| KEY_SEQ | Smallint not NULL | column sequence number in key (starting with 1). |
| PK_NAME | Varchar(128) | primary key identifier. |

***Examples:***

```
HSTMT hstmt;/* statement handle */
SDWORD  NumBytes;/* bytes in a return value */
RETCODE rc;   /* ODBC function return code */

rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
  TableQualifier, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
  TableOwner, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 3, SQL_C_CHAR,
  TableName, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 4, SQL_C_CHAR,
  ColumnName, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 5, SQL_C_SHORT,
```

```
  &KeySeq, 0, &NumRetBytes  );
rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
  PkName, MAX_NAME_LEN, &NumRetBytes);

rc = SQLPrimaryKeys(hstmt, "POLITE", SQL_NTS,
  "OOT_SCH", SQL_NTS, "EMPLOYEE", SQL_NTS);
for(;;) {
  rc = SQLFetch(hstmt);
  if(rc != SQL_SUCCESS) break;
  /* process the information */
}
```

## SQLPutData

***Description:*** Allows an application to pass in data for a parameter or column to the driver at statement execution time. This function can be used to pass in character data values incrementally to a `LONG VARCHAR` column.

***Syntax:***

```
RETCODE SQLPutData(HSTMT hstmt, PTR rgbValue, SWORD cbValue);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`rgbValue`
a pointer to a buffer for passing in data for a parameter or column. The data must use the C data type specified in the Ctype argument of a corresponding SQLBindParameter call. In the current release, the date type must be SQL_C_CHAR.

`cbValue`
the length of data in rgbValue. The amount of data can vary with each call for a given parameter or column.

***Comments:*** See `SQLBindParameter` for an explanation of how data-at-execution parameter data is passed at statement execution time.

***Examples:*** See `SQLParamData`.

## SQLRowCount

***Description:*** Returns the number of rows affected by an `UPDATE`, `INSERT`, or `DELETE` statement associated with a statement handle.

***Syntax:***

```
RETCODE SQLRowCount(HSTMT hstmt, SDWORD *pcrow);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`pcrow`
the number of rows affected by an `UPDATE`, `INSERT`, or `DELETE` statement. This argument is set to -1 if the number of affected rows is not available.

***Examples:***

```
HSTMT hstmt;/* statement handle */
SDWORD  RowCount; /* # of rows */


SQLExecDirect(hstmt, "UPDATE EMPLOYEE SET FIRST_NAME =\
  'DORIS' WHERE FIRST_NAME = 'Doris'", SQL_NTS);
SQLRowCount(hstmt, &RowCount);
```

## SQLSetConnectOption

***Description:*** Sets options that govern aspects of connections.

***Syntax:***

```
RETCODE SQLSetConnectOption(HDBC hdbc, UWORD fOption,
    UWORD vParam);
```

***Key words and parameters:***

`hdbc`
a connection handle.

`fOption`
option to set.

`vParam`
a value associated with fOption. Depending on the value of fOption, vParam is a 32-bit integer value or a pointer to a null-terminated character string.

***Comments:*** All connection options successfully set by an application remain set until `SQLFree-Connect` is called.

The driver supports the following options and values:

| fOption | Valid vParam Values | Description |
|---|---|---|
| SQL_ACCESS_ MODE | SQL_MODE_READ_ONLY<br>SQL_MODE_READ_ WRITE | Indicates the mode of connection. This is used solely for the purposes of optimization. Updates are permitted even when `SQL_MODE_READ_ONLY` is specified. |

| | | |
|---|---|---|
| SQL_ AUTOCOMMIT | SQL_AUTOCOMMIT_ON<br>SQL_AUTOCOMMIT_OFF | If SQL_AUTOCOMMIT_ON is used, the driver commits each statement immediately after it is executed. |
| SQL_CURRENT_ QUALIFIER | null-terminated string | Sets the name of the qualifier to be used by the data source. The qualifier is the name of the database. The database(catalog) must be created using the Personal Oracle Lite CREATDB tool. |
| SQL_TXN_ ISOLATION | SQL_TXN_SERIALIZABLE | Sets the transaction isolation level. |

***Examples:***

```
HENV  henv;/* environment handle */
HDBC  hdbc;/* connection handle */
HSTMT hstmt;/* statement handle */
RETCODE rc;   /* ODBC function return code */

SQLAllocConnect(henv, &hdbc);
rc = SQLConnect(hdbc, "OOT", SQL_NTS,
    "DORIS", SQL_NTS, "OBJECT", SQL_NTS);
if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO) {
  SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF);
  rc = SQLAllocStmt(hdbc,&hstmt);
  if (rc == SQL_SUCCESS) {
    SQLFreeStmt(hstmt, SQL_DROP);
  }
  SQLDisconnect(hdbc);
}
SQLFreeConnect(hdbc);
```

## SQLSetCursorName

***Description:*** Associates a cursor name with a statement handle. If the name is not set by the application, the driver generates a cursor name upon the execution of a select statement.

***Syntax:***

```
RETCODE SQLSetCursorName(HSTMT hstmt,
    UCHAR FAR * szCursor, SWORD cbCursor);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`szCursor`
a cursor name.

`cbCursor`
the length of szCursor.

***Comments:*** The maximum length of a cursor name is 18 characters. A cursor name can be used in positioned update or delete statements. A cursor name that is set remains set until the statement is dropped, using `SQLFreeStmt` with the `SQL_DROP` option.

***Examples:***

```
HSTMT     hstmtSelect;/* select statement handle */
HSTMT     hstmtUpdate;/* update statement handle */
UCHAR     Name[NAME_LEN];
SWORD     Age;
SDWORD    NumBytes;
RETCODE   rc;   /* ODBC function return code */

/* Allocate the statements and set a cursor name */
SQLAllocStmt(hdbc, &hstmtSelect);
SQLAllocStmt(hdbc, &hstmtUpdate);
SQLSetCursorName(hstmtSelect, "EMP_CUR", SQL_NTS);

/* SELECT result set and bind its columns to local variables */
SQLExecDirect(hstmtSelect,
  "SELECT NAME, AGE FROM EMPLOYEE FOR UPDATE", SQL_NTS);
SQLBindCol(hstmtSelect, 1, SQL_C_CHAR,
  Name, NAME_LEN, &NumBytes);
SQLBindCol(hstmtSelect, 2, SQL_C_SHORT, &sAge, 0, &NumBytes);

/* read through the result set until the cursor is
   positioned on the row for Doris */
do {
  rc = SQLFetch(hstmtSelect);
} while ((rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO) &&
(strcmp(Name,"Doris") != 0));
```

```
/* perform a positioned update of Doris's Age */
if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)
  SQLExecDirect(hstmtUpdate,
  "UPDATE EMPLOYEE SET AGE = 25 WHERE CURRENT OF EMP_CUR",
   SQL_NTS);
```

### SQLSetPos

***Description:*** SQLSetPos is used to position the cursor in a rowset and allows the application to add, update or delete data in the rowset. Only the SQL_POSITION and SQL_LOCK_NO_CHANGE options are supported.

### SQLSetStmtOption

***Description:*** Sets options associated with a statement.

***Syntax:***

```
RETCODE SQLSetStmtOption(HSTMT hstmt,
  UWORD fOption, UDWORD vParam);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`fOption`
an option to set.

`vParam`
a value for the option to be set.

***Comments:*** Options for a statement remain set until they are changed by another call to `SQLSetStmtOption` or when the statement is released by calling `SQLFreeStmt` with the `SQL_DROP` option. Some statement options support substitution of a similar value if the data source does not support the specified value of `vParam`. In such cases, the driver returns `SQL_SUCCESS_WITH_INFO` and `SQLSTATE 01S02` (option value changed).

The driver supports the following options and values:

| fOption | Valid vParam Values | Description |
|---|---|---|
| SQL_BIND_TYPE | SQL_BIND_BY_ COLUMN | Sets the binding orientation for SQLExtendedFetch. The driver supports Column-wise binding only. |
| SQL_CONCURRENCY | SQL_CONCUR_LOCK | Sets the cursor concurrency. |
| SQL_CURSOR_TYPE | SQL_CURSOR_ FORWARD_ONLY | Specifies the type of the cursor: SQL_CURSOR_STATIC, or SQL_CURSOR_DYNAMIC |
| SQL_MAX_ROWS | default value | Specifies the maximum number of rows returned to the application when a SELECT statement is executed. |
| SQL_RETRIEVE_DATA | SQL_RD_ON | Specifies that SQLExtendedFetch retrieves data after it positions the cursor to the specified location. |
| SQL_ROWSET_SIZE | (vParam > 0) | Specifies the number of rows in the rowset. |

***Examples:*** See `SQLExtendedFetch`.

## `SQLSpecialColumns`

***Description:*** Returns as a result set the information about certain columns in a given table.

***Syntax:***

```
RETCODE SQLSpecialColumns(HSTMT hstmt, UWORD fColType,
    UCHAR FAR *szTableQualifier, SWORD cbTableQualifier,
    UCHAR FAR *szTableOwner, SWORD cbTableOwner,
    UCHAR FAR *szTableName, SWORD cbTableName,
    UWORD fScope, UWORD fNullable);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`fColType`
the type of column to return. Valid values are SQL_BEST_ROWID and SQL_ROWVER.

`szTableQualifier`
a qualifier name.

`cbTableQualifier`
the length of szTableQualifier.

`szTableOwner`
a string search pattern for owner names.

`cbTableOwner`
the length of szTableOwner.
`szTableName`
a string search pattern for table names.
`cbTableName`
the length of szTableName.
`fScope`
the minimum required scope of rowid.
`fNullable`
a flag indicating whether to return special columns that can have a NULL value.

***Comments:*** When the `fColType` argument is `SQL_ROWVER`, the driver returns a result set with no rows. When the `fColType` argument is `SQL_BEST_ROWID`, the driver returns information about the psuedo-column, `ROWID`. The column `ROWID` uniquely identifies each row in a table. The scope of the rowid is `SQL_SCOPE_SESSION`.

The result set has the following columns:

| Column Name | Data Type | Comments |
|---|---|---|
| SCOPE | Smallint | actual scope of rowid. |
| COLUMN_NAME | Varchar(128) not NULL | column identifier. |
| DATA_TYPE | Smallint not NULL | SQL data type. |
| TYPE_NAME | Varchar(128) not NULL | data source-dependent data type name. |
| PRECISION | Integer | the precision of the column. |
| LENGTH | Integer | the length of the data transferred. |
| SCALE | Smallint | the scale of the column of the data source. |
| PSEUDO_COLUMN | Smallint | a flag indicating whether the column is a pseudo-column. |

***Examples:***

```
HSTMT hstmt;/* statement handle */
SDWORD  NumBytes;/* bytes in a return value */
RETCODE rc;   /* ODBC function return code */
SWORD Scope, DataType, Scale, PseudoCol;
SDWORD  Precision, Length;
UCHAR   ColumnName[MAX_NAME_LEN], TypeName[MAX_NAME_LEN];

rc = SQLBindCol(hstmt, 1, SQL_C_SHORT, &Scope, 0, &NumRetBytes);
```

```
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
    ColumnName, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 3, SQL_C_SHORT, &DataType, 0, &NumRetBytes);
rc = SQLBindCol(hstmt, 4, SQL_C_CHAR,
    TypeName, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 5, SQL_C_SLONG,
    &Precision, 0, &NumRetBytes);
rc = SQLBindCol(hstmt, 6, SQL_C_SLONG, &Length, 0, &NumRetBytes  );
rc = SQLBindCol(hstmt, 7, SQL_C_SHORT, &Scale, 0, &NumRetBytes  );
rc = SQLBindCol(hstmt, 8, SQL_C_SHORT,
    &PseudoCol, 0, &NumRetBytes);

rc = SQLSpecialColumns(hstmt, SQL_BEST_ROWID,
    "POLITE", SQL_NTS, "OOT_SCH", SQL_NTS,
    "EMPLOYEE", SQL_NTS, SQL_SCOPE_SESSION, SQL_NO_NULLS);
for(;;) {
  rc = SQLFetch(hstmt);
  if(rc != SQL_SUCCESS) break;
  /* process the information */
}
```

## SQLStatistics

***Description:*** Returns as a result set the statistics of a table and its indexes.

***Syntax:***

```
RETCODE SQLPrimaryKeys (HSTMT hstmt,
    UCHAR FAR *szTableQualifier, SWORD cbTableQualifier,
    UCHAR FAR *szTableOwner, SWORD cbTableOwner,
    UCHAR FAR *szTableName, SWORD cbTableName,
    UWORD fUnique, UWORD fAccuracy);
```

***Key words and parameters:***

`hstmt`
a statement handle.

`szTableQualifier`
a qualifier name.

`cbTableQualifier`
the length of szTableQualifier.

`szTableOwner`
name of table owner.

`cbTableOwner`
the length of szTableOwner.

`szTableName`
a string search pattern for table names.

`cbTableName`
the length of szTableName.

`fUnique`
the type of index: SQL_INDEX_UNIQUE or SQL_INDEX_ALL.

`fAccuracy`
the importance of the CARDINALITY and PAGES columns in the result set: SQL_ENSURE indicates that the statistics must be accurate. SQL_QUICK indicates that the statistics can be retrieved if they are readily available.

***Comments:*** `SQLStatistics` returns information about a single table as a standard result set, ordered by `NON_UNIQUE`, `TYPE, INDEX_QUALIFIER`, `INDEX_NAME`, and `SEQ_IN_INDEX`.

The following table lists the columns in the result set:

| Column Name | Data Type | Comments |
|---|---|---|
| TABLE_QUALIFIER | Varchar(128) | table qualifier identifier . |
| TABLE_OWNER | Varchar(128) | table owner identifier. |
| TABLE_NAME | Varchar(128) not NULL | table identifier. |
| NON_UNIQUE | Smallint | indicates whether the index disallows duplicate values: TRUE if the index values can be nonunique; FALSE if the index values must be unique. NULL is returned if TYPE is SQL_TABLE_STAT. |
| INDEX_QUALIFIER | Varchar(128) | the identifier that is used to qualify the index name doing a DROP INDEX; |
| INDEX_NAME | Varchar(128) | index identifier. NULL is returned if TYPE is SQL_TABLE_STAT. |
| TYPE | Smallintnot NULL | type of information being returned: SQL_TABLE_STAT indicates a statistic for the table, SQL_INDEX_OTHER indicates another type of index OOT data source supports B-Tree index. |
| SEQ_IN_INDEX | Smallint | column sequence number in index (starting with 1). NULL is returned if TYPE is SQL_TABLE_STAT. |

| | | |
|---|---|---|
| COLUMN_NAME | Varchar(128) | column identifier. |
| COLLATION | Char(1) | sort sequence for the column: "A" for ascending and "D" for descending. NULL is returned if column sort sequence is not supported by the data source or if TYPE is SQL_TABLE_STAT. |
| CARDINALITY | Integer | cardinality of table or index: number of rows in table if TYPE is SQL_TABLE_STAT, or number of unique values in the index if TYPE is not SQL_TABLE_STAT. |
| PAGES | Integer | number of pages used to store the index or table: number of pages for the table if TYPE is SQL_TABLE_STAT, or number of pages for the index if TYPE is not SQL_TABLE_STAT. |
| FILTER_CONDITION | Varchar(128) | null is returned as the filter condition cannot be determined. |

***Examples:*** See `SQLPrimaryKeys`.

## `SQLTablePrivileges`

***Description:*** Returns as a result set the information about the privileges associated with the specified tables.

***Syntax:***

```
RETCODE SQLTablePrivileges(HSTMT hstmt,
    UCHAR FAR *szTableQualifier, SWORD cbTableQualifier,
    UCHAR FAR *szTableOwner, SWORD cbTableOwner,
    UCHAR FAR *szTableName, SWORD cbTableName);
```

***Key words and parameters:***

`hstmt`
a statement handle.
`szTableQualifier`
a qualifier name.
`cbTableQualifier`
the length of szTableQualifier.
`szTableOwner`
a string search pattern for owner names.
`cbTableOwner`
the length of szTableOwner.
`szTableName`
a string search pattern for table names.
`cbTableName`
the length of szTableName.

***Comments:*** `SQLTablePrivileges` returns the information as a result set, ordered by `TABLE_QUALIFIER`, `TABLE_OWNER`, `TABLE_NAME`, and `PRIVILEGE`.

The following table lists the columns in the result set:

| Column Name | Data Type | Comments |
|---|---|---|
| TABLE_QUALIFIER | Varchar(128) | table qualifier identifier. |
| TABLE_OWNER | Varchar(128) | table owner identifier. |
| TABLE_NAME | Varchar(128) not NULL | table identifier. |
| GRANTOR | Varchar(128) | identifier of the user who granted the privilege. |
| GRANTEE | Varchar(128) not NULL | identifier of the user to whom the privilege was granted. |
| PRIVILEGE | Varchar(128) not NULL | identifies the table privilege. |
| IS_GRANTABLE | Varchar(3) | indicates whether the grantee is permitted to grant the privilege to other users; "YES", "NO", or NULL if unknown or not applicable to the data source. |

***Examples:***

```
HSTMT hstmt;/* statement handle */
SDWORD  NumBytes;/* bytes in a return value */
UCHAR TableQualifier[MAX_NAME_LEN];
UCHAR TableOwner[MAX_NAME_LEN];
UCHAR TableName[MAX_NAME_LEN];
UCHAR Grantor[MAX_NAME_LEN];
UCHAR Grantee[MAX_NAME_LEN];
UCHAR Privilege[MAX_NAME_LEN];
UCHAR IsGrantable[MAX_NAME_LEN];
RETCODE rc;   /* ODBC function return code */

rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
    TableQualifier, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
    TableOwner, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 3, SQL_C_CHAR,
    TableName, MAX_NAME_LEN, &NumRetBytes);
```

```
rc = SQLBindCol(hstmt, 4, SQL_C_CHAR,
    Grantor, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 5, SQL_C_CHAR,
    Grantee, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 6, SQL_C_CHAR,
    Privilege, MAX_NAME_LEN, &NumRetBytes);
rc = SQLBindCol(hstmt, 7, SQL_C_CHAR,
    IsGrantable, MAX_NAME_LEN, &NumRetBytes);
rc = SQLTablePrivileges(hstmt,"POLITE", SQL_NTS,
    "OOT_SCH", SQL_NTS, "EMPLOYEE", SQL_NTS);

for(;;) {
  rc = SQLFetch(hstmt);
  if(rc != SQL_SUCCESS) break;
  /* process the information */
}
```

## SQLTables

***Description:*** Returns as a result set the list of table names stored in a data source.

***Syntax:***

```
RETCODE SQLTables(HSTMT hstmt,
    UCHAR *szTableQualifier, SWORD cbTableQualifier,
    UCHAR *szTableOwner, SWORD cbTableOwner,
    UCHAR *szTableName, SWORD cbTableName,
    UCHAR *szTableType, SWORD cbTableType);
```

***Key words and parameters:***

`hstmt`
  a statement handle.
`szTableQualifier`
  a qualifier name.
`cbTableQualifier`
  the length of szTableQualifier.
`szTableOwner`
  a string search pattern for owner names.
`cbTableOwner`
  the length of szTableOwner.
`szTableName`

a string search pattern for table names.

`cbTableName`
the length of szTableName.

`szTableType`
a list of table types to match.

`cbTableType`
the length of szTableType list.

***Comments:*** `SQLTables` returns as a standard result set all the tables found in the requested range.

The OOT ODBC driver currently supports the following special semantics:

If `szTableType` uses a single percent character (%) while `szTableQualifier`, `szTableOwner`, and `szTableName` are all empty strings, then the result set is a list of valid data types in the given search range. (All columns except the `TABLE_TYPE` column are `NULL`.)

If `szTableType` is not an empty string, it must contain a list of comma-separated type name strings of interest; each such string can be enclosed in single quotes (‘’) or unquoted. The result set is ordered on the `TABLE_NAME` column.

***Examples:***

```
#define NAME_LEN128
HSTMT hstmt;  /* statement handle */
char  szName[NAME_LEN];/* name buffer */
RETCODE rc;       /* ODBC function return code */

/* retrieve all names of tables owned by "DORIS"
   in the catalog OOT_SCH ... */
SQLTables(hstmt, "POLITE, SQL_NTS, "DORIS", SQL_NTS,
        NULL, 0, "TABLE", SQL_NTS);
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA_FOUND) {
  SQLGetData(hstmt,3, SQL_C_CHAR, szName, NAME_LEN, NULL);
  /* do something with the table name in szName ... */
}
```

## SQLTransact

***Description:*** Requests a commit or rollback operation for all active operations on all statement handles associated with a connection. An application can also call `SQLTransact` to request that a commit or rollback operation be performed for all connections associated

with an environment handle.

***Syntax:***

```
RETCODE QLTransact(HENV henv, HDBC hbdc, UWORD fType);
```

***Key words and parameters:***

`henv`
an environment handle.

`hdbc`
a connection handle.

`fType`
SQL_COMMIT or SQL_ROLLBACK.

***Comments:*** If the requested transaction is successfully completed, the OOT ODBC driver closes all open cursors and leaves all statement handles in the prepared state.

***Examples:*** See `SQLParamData`.

CHAPTER

7

# ODBC Information Types

This chapter describes and lists the ODBC information types that can be queried using the SQLGetInfo function.

| ODBC Information Type | Description |
|---|---|
| SQL_ACCESSIBLE_TABLES | (There are tables that can't be accessed) |
| SQL_ACCESSIBLE_PROCEDURES | (not applicable) |
| SQL_ACTIVE_CONNECTIONS | (unknown) |
| SQL_ACTIVE_STATEMENTS | (unknown) |
| SQL_ALTER_TABLE | (not applicable) |
| SQL_BOOKMARK_PERSISTENT | (not applicable) |
| SQL_COLUMN_ALIAS | "Y" |
| SQL_CONCAT_NULL_BEHAVIOR | SQL_CB_NULL=0 |
| SQL_CONVERT_BIGINT | (not supported) |
| SQL_CONVERT_BINARY | (not supported) |
| SQL_CONVERT_BIT | (not supported) |
| SQL_CONVERT_CHAR | (not supported) |
| SQL_CONVERT_DATE | (not supported) |
| SQL_CONVERT_DECIMAL | (not supported) |
| SQL_CONVERT_DOUBLE | (not supported) |
| SQL_CONVERT_FLOAT | (not supported) |
| SQL_CONVERT_FUNCTIONS | (not supported) |
| SQL_CONVERT_INTEGER | (not supported) |
| SQL_CONVERT_ LONGVARBINARY | (not supported) |
| SQL_CONVERT_LONGVARCHAR | (not supported) |
| SQL_CONVERT_NUMERIC | (not supported) |
| SQL_CONVERT_REAL | (not supported) |
| SQL_CONVERT_SMALLINT | (not supported) |

| ODBC Information Type | Description |
|---|---|
| SQL_CONVERT_TIME | (not supported) |
| SQL_CONVERT_TIMESTAMP | (not supported) |
| SQL_CONVERT_TINYINT | (not supported) |
| SQL_CONVERT_VARBINARY | (not supported) |
| SQL_CONVERT_VARCHAR | (not supported) |
| SQL_CORRELATION_NAME | SQL_CN_DIFFERENT=1 |
| SQL_CURSOR_COMMIT_ BEHAVIOR | SQL_CB_CLOSE=1 |
| SQL_CURSOR_ROLLBACK_ BEHAVIOR | SQL_CB_CLOSE=1 |
| SQL_DATA_SOURCE_READ_ONLY | "N" |
| SQL_DATABASE_NAME | (not applicable) |
| SQL_DBMS_NAME | "ORDBMS 2.3" |
| SQL_DBMS_VER | "POLITE ORDBMS V02.03.00xx" |
| SQL_DEFAULT_TXN_ISOLATION | SQL_TXN_READ_UNCOMMITTED |
| SQL_DRIVER_NAME | "OOTOD32.DLL" |
| SQL_DRIVER_ODBC_VER | "02.01" |
| SQL_DRIVER_VER | "01.00.0000" |
| SQL_EXPRESSIONS_IN_ORDERBY | (not supported) |
| SQL_FETCH_DIRECTION | SQL_FD_FETCH_NEXT |
| SQL_FILE_USAGE | SQL_FILE_NOT_SUPPORTED=0 |
| SQL_GETDATA_EXTENSIONS | SQL_GD_ANY_COLUMN \| SQL_GD_ANY_ORDER |
| SQL_GROUP_BY | SQL_GB_NOT_SUPPORTED=0 |

| ODBC Information Type | Description |
|---|---|
| SQL_KEY WORDS | "BIGINT,CONCAT,DAYOFMONTH, DAYOFWEEK,DAYOFYEAR,INT,READ, REAL,SESSION,TIMESTAMPADD, TIMESTAMPDIFF,WRITE,ZONE" |
| SQL_IDENTIFIER_CASE | SQL_IC_UPPER=1 |
| SQL_IDENTIFIER_QUOTE_CHAR | """ |
| SQL_LIKE_ESCAPE_CLAUSE | "Y" |
| SQL_LOCK_TYPES | (not applicable) |
| SQL_MAX_BINARY_LITERAL_LEN | (not applicable) |
| SQL_MAX_CHAR_LITERAL_LEN | 1023 |
| SQL_MAX_COLUMN_NAME_LEN | 50 |
| SQL_MAX_COLUMNS_IN_GROUP_ BY | 1 |
| SQL_MAX_COLUMNS_IN_INDEX | 1 |
| SQL_MAX_COLUMNS_IN_ORDER_ BY | 1 |
| SQL_MAX_COLUMNS_IN_SELECT | (unknown) |
| SQL_MAX_COLUMNS_IN_TABLE | (unknown) |
| SQL_MAX_CURSOR_NAME_LEN | (not applicable) |
| SQL_MAX_INDEX_SIZE | (unknown) |
| SQL_MAX_OWNER_NAME_LEN | 8 |
| SQL_MAX_PROCEDURE_NAME_ LEN | (not applicable) |
| SQL_MAX_QUALIFIER_NAME_ LEN | (not applicable) |
| SQL_MAX_ROW_SIZE | (unknown) |
| SQL_MAX_ROW_SIZE_INCLUDES_ LONG | (not applicable) |

| ODBC Information Type | Description |
|---|---|
| SQL_MAX_STATEMENT_LEN | (unknown) |
| SQL_MAX_TABLE_NAME_LEN | (unknown) |
| SQL_MAX_TABLES_IN_SELECT | (unknown) |
| SQL_MAX_USER_NAME_LEN | 128 |
| SQL_MULT_RESULT_SETS | (not supported) |
| SQL_MULTIPLE_ACTIVE_TXN | (not applicable) |
| SQL_NEED_LONG_DATA_LEN | "Y" |
| SQL_NON_NULLABLE_COLUMNS | SQL_NNC_NON_NULL=1 |
| SQL_NULL_COLLATION | SQL_NC_LOW=1 |
| SQL_NUMERIC_FUNCTIONS | (not supported) |
| SQL_OUTER_JOINS | (not supported) |
| SQL_OWNER_TERM | "Schema" |
| SQL_ODBC_API_CONFORMANCE | SQL_OAC_LEVEL1=1 |
| SQL_ODBC_SAG_CLI_ CONFORMANCE | SQL_OSCC_NOT_COMPLIANT=0 |
| SQL_ODBC_SQL_CONFORMANCE | SQL_OSC_MINIMUM=0 |
| SQL_ODBC_SQL_OPT_IEF | (not supported) |
| SQL_ODBC_VER | "02.00.0000" |
| SQL_ORDER_BY_COLUMNS_IN_ SELECT | "Y" |
| SQL_OWNER_USAGE | SQL_OU_DML_STATEMENTS \| SQL_OU_TABLE_DEFINITION |
| SQL_POS_OPERATIONS | (not supported) |
| SQL_POSITIONED_STATEMENT | (not supported) |
| SQL_PROCEDURE_TERM | (not applicable) |
| SQL_PROCEDURES | (not applicable) |

| ODBC Information Type | Description |
|---|---|
| SQL_QUALIFIER_NAME_ SEPARATOR | (not applicable) |
| SQL_QUALIFIER_LOCATION | SQL_QL_START=1 |
| SQL_QUALIFIER_TERM | "SCHEMA" |
| SQL_QUALIFIER_USAGE | (not applicable) |
| SQL_QUOTED_IDENTIFIER_CASE | SQL_IC_SENSITIVE=3 |
| SQL_ROW_UPDATES | (not applicable) |
| SQL_SCROLL_CONCURRENCY | SQL_SCCO_READ_ONLY |
| SQL_SCROLL_OPTIONS | SQL_SO_FORWARD_ONLY |
| SQL_SEARCH_PATTERN_ESCAPE | (not supported) |
| SQL_SERVER_NAME | (not applicable) |
| SQL_SPECIAL_CHARACTERS | (no special characters in object name) |
| SQL_STATIC_SENSITIVITY | (not applicable) |
| SQL_STRING_FUNCTIONS | SQL_FN_STR_CONCAT \| SQL_FN_STR_LTRIM \| SQL_FN_STR_LENGTH \| SQL_FN_STR_LOCATE \| SQL_FN_STR_LCASE \| SQL_FN_STR_RTRIM \| SQL_FN_STR_SUBSTRING \| SQL_FN_STR_UCASE \| SQL_FN_STR_LOCATE_2 |
| SQL_SUBQUERIES | (not supported) |
| SQL_SYSTEM_FUNCTIONS | (not supported) |
| SQL_TABLE_TERM | "table" |
| SQL_TIMEDATE_ADD_INTERVALS | (not applicable) |
| SQL_TIMEDATE_DIFF_INTERVALS | (not applicable) |

| ODBC Information Type | Description |
|---|---|
| SQL_TIMEDATE_FUNCTIONS | SQL_FN_TD_NOW \|<br>SQL_FN_TD_CURDATE \|<br>SQL_FN_TD_DAYOFMONTH \|<br>SQL_FN_TD_DAYOFWEEK \|<br>SQL_FN_TD_DAYOFYEAR \|<br>SQL_FN_TD_MONTH \|<br>SQL_FN_TD_YEAR \|<br>SQL_FN_TD_CURTIME \|<br>SQL_FN_TD_HOUR \|<br>SQL_FN_TD_MINUTE \|<br>SQL_FN_TD_SECOND |
| SQL_TXN_CAPABLE | SQL_TC_ALL=2 |
| SQL_TXN_ISOLATION_OPTION | SQL_TXN_READ_UNCOMMITTED |
| SQL_UNION | (not supported) |
| SQL_USER_NAME | (not supported) |

CHAPTER

# 8

# System Catalog

This chapter describes the following objects in the system catalog:

- ALL_USERS
- ALL_CONSTRAINTS
- ALL_CONS_COLUMNS
- ALL_TABLES
- ALL_TAB_COLUMNS
- ALL_VIEWS
- COLUMN_PRIVILEGES
- DUAL
- TABLE_PRIVILEGES
- USER_OBJECTS

**Note:** Columns marked with an asterisk are provided for compatibility with the Oracle7 catalog views and generally return NULL or a default value.

## ALL_USERS

This object provides the following information about all schemas created in the connected database:

***USERNAME***

name of the schema

***USER_ID***

ID number of the schema

***CREATED***

schema creation data

## ALL_CONSTRAINTS

This object provides the following information about constraint definitions on accessible tables:

***OWNER***

owner of the constraint definition

***CONSTRAINT_NAME***

name associated with the constraint definition

***CONSTRAINT_TYPE***

type of constraint definition:

C (check constraint on a table),
P (primary key),
U (unique key),
R (referential integrity), or
V (with check option, on a view)

***TABLE_NAME***

name of table with constraint definition

***SEARCH_CONDITION***

text of search condition for table check

***R_OWNER***

owner of table used in referential constraint

***R_CONSTRAINT_NAME***

name of unique constraint definition for referenced table

***DELETE_RULE***

delete rule for a referential constraint: NO ACTION

***STATUS***

status of constraint: ENABLED or DISABLED

## ALL_CONS_COLUMNS

This object provides the following information about accessible columns in constraint definitions:

***OWNER***

username of owner of the constraint definition

***CONSTRAINT_NAME***

name associated with the constraint definition

***TABLE_NAME***

name of table with constraint definition

***COLUMN_NAME***

name associated with column specified in the constraint definition

***POSITION***

original position of column in definition

## ALL_TABLES

This object provides the following information about tables accessible to the user:

***OWNER***

username of owner of the table

***TABLE_NAME***

name of the table

***TABLESPACE_NAME***

name of the catalog or database file containing the table

***CLUSTER_NAME****

name of the cluster, if any, to which the table belongs

***PCT_FREE****

minimum percentage of free space in a block

***PCT_USED****

minimum percentage of used space in a block

***INI_TRANS****

initial number of transactions

***MAX_TRANS****

maximum number of transactions

***INITIAL_EXTENT****

size of the initial extent in bytes

***NEXT_EXTENT****

size of secondary extents in bytes

***MIN_EXTENTS****

minimum number of extents allowed in the segment

***MAX_EXTENTS****

maximum number of extents allowed in the segment

***PCT_INCREASE****

percentage increase in extent size

***BACKED_UP****

has table been backed up since last change

***NUM_ROWS****

number of rows in the table

***BLOCKS****

number of data blocks allocated to the table

***EMPTY_BLOCKS****

number of data blocks allocated to the table that contain no data

***AVG_SPACE****

average amount of free space (in bytes) in a data block allocated to the table

***CHAIN_CNT****

number of rows in the table that are chained from one data block to another, or that have migrated to a new block, requiring a link to preserve the old rowid

***AVG_ROW_LEN****

average length of a row in the table in bytes

## ALL_TAB_COLUMNS

This object provides the following information about the columns of tables, views, and clusters accessible to the user:

***OWNER***

username of owner of the table, view, or cluster

***TABLE_NAME***

table, view, or cluster name

***COLUMN_NAME***

column name

***DATA_TYPE***

datatype of the column

***DATA_LENGTH***

length of the column in bytes

***DATA_PRECISION***

decimal precision for NUMERIC and DECIMAL data type; binary precision for FLOAT, REAL and DOUBLE data type; NULL for all other data types

***DATA_SCALE***

digits to right of decimal point in a NUMERIC or DECIMAL

***NULLABLE***

does column allow NULLs? Value is N if there is a NOT NULL constraint on the column or if the column is part of a PRIMARY KEY

***COLUMN_ID***

sequence number of the column as created

***DEFAULT_LENGTH***

length of default value for the column

***DATA_DEFAULT***

default value for the column

***NUM_DISTINCT****

number of distinct values in each column of the table

***LOW_VALUE****

***HIGH_VALUE****

for tables with more than three rows, the second-lowest and second-highest values in the column. For tables with three rows or fewer, the lowest and highest values. These statistics are expressed in hexadecimal notation for the internal representation of the first 32 bytes of the values

## ALL_VIEWS

This object provides the following information about views accessible to the user:

***OWNER***

username of the owner of the view

***VIEW_NAME***

name of the view

***TEXT_LENGTH***

length of the view text

***TEXT***

view text

## CAT

This object provides the following information about tables and views accessible to the user:

***TABLE_NAME***

name of the object

***TABLE_TYPE***

type of the object

## COLUMN_PRIVILEGES

This object provides the following information about grants on columns for which the user is the grantor, grantee, or owner, or, for which PUBLIC is the grantee:

***OWNER***

username of the owner of the object

***TABLE_NAME***

name of the object

***COLUMN_NAME***

name of the column

***GRANTOR***

name of the user who performed the grant

***GRANTEE***

name of the user to whom access was granted

***GRANT_TYPE***

privilege on the object. The value can be SELECT, INSERT, or DELETE

***GRANTABLE***

YES if the privilege was granted with GRANT OPTION, otherwise NO

## DUAL

This object is a dummy table which can be used in a query when you want to return a single row. For example, you could use DUAL to select CURRENT_TIMESTAMP.

| ***DUMMY*** | ***ROWNUM*** |
|---|---|
| VARCHAR2 (1) | number (10) |

(always NULL) (always 1)

## TABLE_PRIVILEGES

This object provides the following information about grants on objects for which the user or PUBLIC is the grantee:

***OWNER***

username of the owner of the object

***TABLE_NAME***

name of the object

***GRANTOR***

name of the user who performed the grant

***GRANTEE***

name of the user to whom access is granted

***GRANT_TYPE***

privilege on the object. The value can be one of the following: SELECT, INSERT, or DELETE

***GRANTABLE***

YES if the privilege was granted with GRANT OPTION, otherwise NO

## USER_OBJECTS

This object provides the following information about objects accessible to the user:

***OWNER***

username of the owner of the object

***OBJECT_NAME***

name of the object

***OBJECT_ID***

object identifier of the object

***OBJECT_TYPE***

type of the object

***CREATED***

timestamp for the creation of the object

***LAST_DDL_TIME***

timestamp for the last modification of the object resulting from a DDL command (including grants and revokes)

***CREATED_TIME***

timestamp for the creation of the object (character data)

***STATUS****

status of the object: VALID, INVALID, or N/A

(always valid)

CHAPTER

# 9 Database Utilities

This chapter describes how to use the following database utilities:

- CREATEDB, for creating Personal Oracle Lite databases
- REMOVEDB, for removing Personal Oracle Lite databases
- ODBC Administrator, for managing ODBC connections
- POLDM16, for managing databases

## CREATEDB

***Description:*** Utility for creating a database.

***Syntax:***

```
CREATEDB DSN_Name Database_Name [Database_Id Database_Size Extent_Size]
```

***Key words and parameters:***

***DSN_Name***

Data source name, used to look up the ODBC.INI file for the default database directory and unique Database_Id.

***Database_Name***

Name of the database to be created. It can be a full path name or just the database name. If only the database name is given, the database is created under the Data Directory for the DSN specified in the ODBC.INI file. The extension for the database name must always be ".odb". If a name without the ".odb" is given, the ".odb" is appended.

***Database_Id***

(Optional) When specified, it is used as the Database_Id (instead of getting the Database_Id from the ODBC.INI file).

**Note:** The ID must be unique for each database in a directory.

***Database_Size***

(Optional) Maximum size for the database to be created. Default size is 256M.

***Extent_Size***

(Optional) Number of pages per extent. Default is 4. The extent is the unit of allocation for each table.

***Examples:***

```
createdb polite db1
createdb polite c:\testdir\db2.odb 300
createdb polite db3 301 10000
createdb polite db4 500 256000 2
```

## REMOVEDB

***Description:*** Utility for deleting a database.

***Syntax:***

```
REMOVEDB DSN_Name Database_Name
```

***Key words and parameters:***

`<DSN_Name>`

Data source name, used to look up the ODBC.INI file for the default database directory.

`<Database_Name>`

Name of the database to be deleted. It can be a full path name or just the database name. If only the database name is given, the database is deleted from the Data Directory as specified in the ODBC.INI file (for the DSN).

***Examples:***

```
removedb polite db1
removedb polite c:\testdir\db2.obd
```

## ODBC ADMINISTRATOR

***Description:*** A utility provided by Microsoft to manage the ODBC.INI file and associated registry entries in Windows 3.1x, Windows 95 and Windows NT. It allows you to add a data source name and specify the database file you want to dedicate as the default for the data source name.

***Syntax:*** Select the ODBC Administrator entry in the Oracle for Windows program group, or enter one of the following commands on the command line:

- ODBCAD32 in Windows 95 or Windows NT
- ODBCADM in Windows 3.1x

***Comments:*** The default ODBC data source name, POLITE, is created when you install Personal Oracle Lite. However, you can create additional data source names to which you can assign Personal Oracle Lite database files you create. For example, to add a new data source name,`MYDSN`, and dedicate to it a default database `MYDB`, follow these steps:

1. Click on ADD.
2. Double-click on the Oracle Lite ODBC driver from the list of Installed ODBC Drivers.

   The Personal Oracle Lite ODBC Drive Setup dialog box appears.
3. Set the data source name to `MYDSN`. You may provide a description.
4. Type in the path to the data directory where the database files should reside, for example, in C:\ORAWIN\POLDB.

   **Note:** If the path you specify does not exist, you will not be able to exit the dialog box in the usual way.

5. Give the name of the database file you want to dedicate as the default for the data source name, for example `MYDB`.

   **Note:** Do not include the .ODB extension.

6. Click on OK. A new data source named `MYDSN` is created, with a dedicated database named `MYDB`.

## POLDM16

***Description:*** A utility for managing Personal Oracle Lite databases, intended for use in 16-bit environments. Data Manager (POLDM16.EXE) performs a subset of functions available in the Navigator, which is available only in 32-bit environments.

You can use Data Manager to transfer data between two databases, back up and restore database files, export and import delimited data, and load data from SQL*Loader files.

***Comments:*** The Data Manager cannot transfer any single data (a column data in a particular row of the table) that exceeds 64K between two databases.

APPENDIX

# A

# Differences Between Oracle7 and Personal Oracle Lite

This chapter describes key differences in behavior between Personal Oracle Lite and Oracle 7.

## COMMIT and ROLLBACK

Oracle7 COMMITs after every DDL statement. Personal Oracle Lite does not until you explicitly issue the COMMIT command. For example:

```
INSERT INTO EMP VALUES ('JAMESON', 7456, 20...);
CREATE TABLE DEPT (LOC NUMBER, CITY CHAR(10));
ROLLBACK;
```

In Oracle7, the creation of `DEPT` table will force the `COMMIT` of the `INSERT`. In Personal Oracle Lite, both the `INSERT` and the creation of the `DEPT` table will rollback.

Similarly:

```
CREATE TABLE DEPT (LOC NUMBER, CITY CHAR(10));
INSERT INTO DEPT VALUES( 10, 'LOS ANGELES');
INSERT INTO DEPT VALUES( 20, 'BOSTON');
ROLLBACK;
```

For this example, in Oracle7 the `DEPT` table will exist but will be empty. In Personal Oracle Lite, the `DEPT` table will not exist.

## Creating Tables

```
CREATE TABLE <newtablename> AS SELECT * FROM <oldtablename>
```

This syntax is not supported by Personal Oracle Lite. You must create the table using the appropriate DDL, `COMMIT` the DDL, then you may copy the existing table into the new table using:

```
INSERT INTO newtable SELECT * FROM oldtable;
```

## Indicator Variables

Indicator variables are 32-bit LONG integers in Personal Oracle Lite. In Oracle7, 16-bit SHORT integers are used.

## Computed Columns

The following statement:

```
deptno, SUM(sal) FROM emp GROUP BY deptno;
```

displays the SUM() column heading as "OOT_SCH x x". You should always provide an alias for the name of computed columns, as in the following example:

```
SELECT deptno, SUM(sal) TotalSalary FROM emp GROUP BY deptno;
```

### Data Precision During Arithmetic Operations

Oracle7 looks at the datatype on the left side of an assignment when deciding how many decimal places of a result to store into a column. Personal Oracle Lite follows SQL-92 convention, and only provides the maximum number of digits of precision from the right side of the assignment.

### Tables not Installed with Personal Oracle Lite

The table `system.product_privs`, which contains product user profiles in an Oracle 7 databse, is not installed with Personal Oracle Lite.

The Standard Oracle demo-tables EMP, DEPT, BONUS, and SALGRADE as well as DEMOEMP and DEMODEPT are generated by the POBLD.SQL script.

### Messages

Personal Oracle Lite does not generate the same messages in response to SQL*Plus commands as other Oracle databases generate. For example, if you update a Persoal Oracle Lite table, the message 'Operation 0 succeeded' is returned, no matter how many records are updated, and even if no records are updated. When running SQL*Plus against other Oracle databases you get a message back telling you how many records were updated.

### Sequences

Personal Oracle Lite does not support 'CYCLE' commands in sequence statements. Sequence numbers are also subject to ROLLBACK in version 2.3.

APPENDIX

# B

# Database Parameters in POLITE.INI

You can customize your Personal Oracle Lite databases by changing the database parameter values defined in your POLITE.INI file. This chapter discusses the POLITE.INI file and its associated parameters.

## About the POLITE.INI File

The POLITE.INI file centralizes database volume ID assignments and defines parameters for all databases on a system. When you install Personal Oracle Lite, the Oracle Installer creates POLITE.INI in your \Windows, \Win95, or \WinNt directory.

The Oracle Installer automatically sets the parameters in your POLITE.INI file, but you can modify them to customize the product's behavior. To modify the POLITE.INI file, you can use an ASCII text editor, such as Notepad.

## POLITE.INI Parameters

The POLITE.INI file contains one section, "All Databases," under which are listed the the parameters described in the table below.

| Parameter | Description |
|---|---|
| DatabaseID=n | Defines the next database volume ID number to be assigned by either the CREATEDB program or the CREATE DATABASE SQL command. Database volume ID numbers must be unique for each database file on the system. |
| CacheSize=n | Specifies the size of the object cache in kilobytes. The minimum is 128. If not set, the default is 4096 (4 megabytes). |
| MessageFile=path | Specifies the path to the binary error message file, POLITEUS.MSB. The Oracle Installer sets this parameter to the full path for the error message file. |
| TempDir=path | Specifies the directory where the temporary database POLTEMP.ODB will be created. If not set, the default is any TEMP, TMP, or WINDIR setting defined in your environment. |

## Sample POLITE.INI File

The following is a sample POLITE.INI file:

```
[All Databases]

DatabaseID=128
CacheSize=4096
MessageFile=C:\ORAWIN95\DBS\POLITEUS.MSB
TempDir=D:\TMP
```

# Index

# D

## E

# F



# G



# H



# I

## J



## L



## M



## N



## O

# P

# Q



# R

# S

## T



## U



## V



## W

# Reader's Comment Form

## Name of Document: Personal Oracle Lite™ Programmer's Guide A52614-02

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?
- Do you better access to the information that you need? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:


Oracle Workgroup Products Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

or e-mail comments to: dlahey@us.oracle.com


If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.