Class Note: Chapter 4

Basic Database Access with SQL

(Updated on March 2009 based upon Oracle 11g, on May 12, 2016, 12c June 25, 2018, and 18c May 2019)

[The “class note” is the typical material I would prepare for my face-to-face class. Since this is an Internet based class, I am sharing the notes with everyone assuming you are in the class.]

Please note this class/chapter is the most essential in learning Oracle. This class note is organized in a tutorial format. Students are urged to follow the exercises in chapter, step by step, in order to quickly learn the concept and operation. I would highly recommend that you retype the SQL command in your computer as doing so reinforce the learning and understanding. You will be amazed by yourself if you can do those exercises as described in this class.

To get work done, applications must communicate with Oracle to enter and retrieve data, and do so in a way that protects the integrity of the database's data. This chapter introduces the basic concepts of how applications use SQL statements and encompassing transactions to interact with an Oracle database system.

Chapter Prerequisites

Support Files for Hands-On Exercises in This Course

All subsequent chapters for this course contain hands-on exercises that provide you with invaluable experience using Oracle database server. At the beginning of most chapters, a section called "Chapter Prerequisites" explains the steps necessary to prepare for the chapter's exercises. Typically, you'll be asked to use Oracle's SQL*Plus utility to run a SQL command script that builds a practice schema for the chapter, complete with a set of tables and other supporting database objects. Most scripts also set up custom display formats for the SQL*Plus session, so that the results produced by the chapter's exercises are easy to read.

CAUTION

*Unless otherwise indicated, after you successfully run a chapter's supporting SQL command script using SQL*Plus, do not exit the SQL*Plus session. You should use the SQL *Plus session to complete the exercises in the chapter, starting with the first* *exercise in the chapter, all the way through the final exercise in the chapter. If you do not have time to complete all exercises in the chapter during one sitting, leave your computer running and the SQL*Plus session open so that you can pick up where you left off when you have more time. If you must shut down your computer, the next time that you start SQL*Plus you must rerun the chapter's SQL command script (to refresh the necessary data and SQL*Plus display settings) and then repeat all exercises in the chapter, starting with the first exercise again.*

You will need to download the supplemental files, and then unzip (extract) them into a location in your own computer. To download, either go the schedule page of the course or directly copy the following link to your browser:

http://www.eiu.edu/~pingliu/tec5323/Resources/tec5323_code.zip

From now on, we assume you have already downloaded and unzipped your supplemental files.

To practice the hands-on exercises in this chapter, you need to start SQL*Plus and run the following command script at SQL > prompt:

```
SQL>location\Sql\chap04.sql
```

Where location is the file directory where you expanded the support archive that accompanies this book. For example, after starting SQL*Plus and connecting as SCOTT/tiger, you can run this chapter's SQL command script using the SQL*Plus command @, as in the following example (assuming that your chap04.sql file is in "C:\temp\Sql" folder):

```
SQL> @C:\temp\Sql\chap04.sql;
```

Follow the instructions on screen. The script will ask you for the password of user "SYSTEM." This password is the same as what you entered while you installed Oracle database on your computer. Once the script completes successfully, leave the current SQL*Plus session open and use it to perform this chapter's exercises in the order that they appear.

NOTE:

This is a typical way of developing and deploying a database design in software projects.

4.1 What Is SQL?

To accomplish work with Oracle, applications (software) must use Structured Query Language (SQL) commands. SQL (pronounced either as "sequel" or "essque-ell") is a relatively simple command language that database administrators, developers, and application developers or application users (program) can use to

- Retrieve, enter, update, and delete database data
- Create, alter, and drop database objects
- Restrict access to database data and system operations

The only way that an application can interact with an Oracle database server is to issue a SQL command. Sophisticated graphical user interfaces might hide SQL commands from users and developers, but under the covers, an application always communicates with Oracle using SQL.

4.1.1 Types of SQL Commands

The four primary categories of SQL commands are DML, transaction control, DDL, and DCL commands.

- Data manipulation or data modification language (DML) commands are SQL commands that retrieve, insert, update, and delete table rows in an Oracle database. The four basic DML commands are SELECT, INSERT, UPDATE, and DELETE. Subsequent sections of this chapter provide you with a thorough introduction to these four commands.

- Applications that use SQL and relational databases perform work by using transactions. A database transaction is a unit of work accomplished by one or more related SQL statements. To preserve the integrity of information in a database, relational databases such as Oracle ensure that all work within each transaction either commits or rolls back. An application uses the transaction control SQL commands COMMIT and ROLLBACK to control the outcome of a database transaction. Subsequent sections of this chapter explain how to design transactions and use transaction control SQL commands.

- Data definition language (DDL) commands create, alter, and drop database objects. Most types of database objects have corresponding CREATE, ALTER, and DROP commands. In Chapter 7, we will have more information about, and examples of, several DDL commands.

- An administrative application uses data control language (DCL) commands to control user access to an Oracle database. The three most commonly used DCL commands are the GRANT, REVOKE, and SET ROLE commands. In Chapter 9, we will discuss more about, and provide examples of, these DCL commands.

4.1.2 Application Portability and the ANSI/ISO SQL Standard

The ANSI/ISO SQL standard defines a generic specification for SQL. Most commercial relational database systems, including Oracle and Microsoft SQL Server, support ANSI/ISO standard SQL. When a database supports the SQL standard and an application uses only standard SQL commands, the application is said to be portable. In other words, if you decide to substitute another database that supports the ANSI/ISO SQL standard, the application continues to function unmodified.

The ANSI/ISO SQL-92 standard has four different levels of compliance: Entry, Transitional, Intermediate, and Full. Oracle complies with the SQL-92 Entry level, and has many features that conform to the Transitional, Intermediate, and Full levels. Oracle also has many features that comply with the SQL3 standard, including its new object-oriented database features.

Oracle also supports many extensions to the ANSI/ISO SQL-92 standard. Such extensions enhance the capabilities of Oracle. SQL extensions can take the form of nonstandard SQL commands or just nonstandard options for standard SQL commands. However, understand that when an application makes use of proprietary Oracle SQL extensions, the application is no longer portable—most likely, you would need to modify and recompile the application before it will work with other database systems.

Now that you have a general understanding of SQL, the remaining sections in this class introduce you to the SQL commands that you will most often use to access an Oracle database: SELECT, INSERT, UPDATE, DELETE, COMMIT, and ROLLBACK.

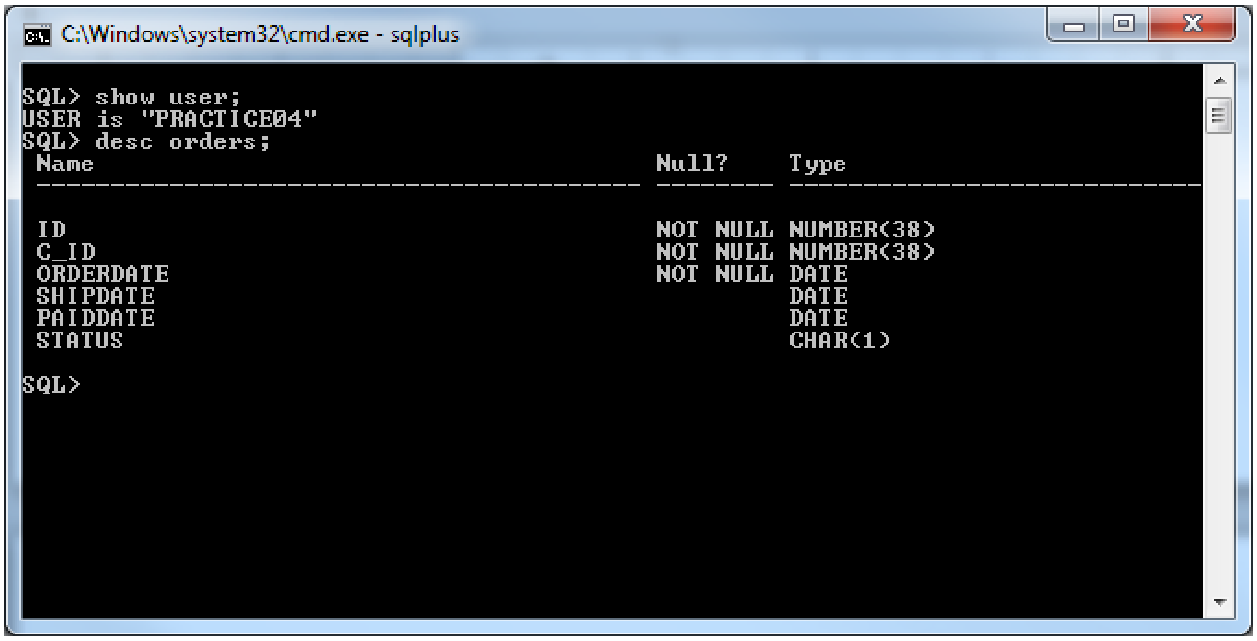
4.2 Retrieving Data with Queries

The most basic SQL statement is a query. A query is a SQL statement that uses the SELECT command to retrieve information from a database. A query's result is the set of columns and rows that the query requests from a database server. For example, the following query retrieves all rows and columns from the ORDERS table:

```
SQL> SELECT * FROM orders;
```

```
   SQL> SELECT * FROM orders;

        ID       C_ID ORDERDATE SHIPDATE  PAIDDATE  S
---------- ---------- --------- --------- --------- -
         1          1 18-JUN-99 18-JUN-99 30-JUN-99 F
         2          2 18-JUN-99                     B
         3          3 18-JUN-99 18-JUN-99 21-JUN-99 F
         4          4 19-JUN-99 21-JUN-99 21-JUN-99 F
         5          5 19-JUN-99 19-JUN-99 28-JUN-99 F
         6          6 19-JUN-99 19-JUN-99           F
         7          7 19-JUN-99                     B
         8          8 20-JUN-99 20-JUN-99 20-JUN-99 F
         9          9 21-JUN-99                     B
        10          2 21-JUN-99 22-JUN-99 22-JUN-99 F
        11          4 22-JUN-99 22-JUN-99           F
        12          7 22-JUN-99 23-JUN-99 30-JUN-99 F
        13          4 22-JUN-99                     B
        14          1 23-JUN-99 25-JUN-99           F

14 rows selected.
```

In order for you to see the above results, you will need to run the chap04.sql script as described at the Chapter Prerequisites. If you did, you would be able to see a similar screen as follows:



```
SQL> show user;
USER is "PRACTICE04"
SQL> desc orders;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------

 ID                                        NOT NULL NUMBER(38)
 C_ID                                      NOT NULL NUMBER(38)
 ORDERDATE                                 NOT NULL DATE
 SHIPDATE                                           DATE
 PAIDDATE                                           DATE
 STATUS                                             CHAR(1)

SQL>
```

4.2.1 The Structure of a Query

Although the structure of a SELECT statement can vary, all queries have two basic components: a SELECT clause and a FROM clause.

A query's SELECT clause specifies a column list that identifies the columns that must appear in the query's result set. Each column in the SELECT clause must correspond to one of the tables in the query's FROM clause. A SELECT clause can also contain expressions that derive information from columns using functions or operators that manipulate table data. Subsequent sections of this chapter explain how to build expressions in a query's SELECT clause.

A query's FROM clause specifies the rows for the query to target. The FROM clause of a typical query specifies a list of one or more tables. Simple queries target just one table, while more advanced queries join information by targeting multiple related tables. Alternatively, a FROM clause in a query can specify a subquery (a nested or inner query) to build a specific set of rows as the target for the main (or outer) query. When you use a subquery in a query's FROM clause, SELECT clause expressions in the outer query must refer to columns in the result set of the subquery.

4.2.2 Building Basic Queries

The following set of hands-on exercises teaches you how to build basic queries and several related functions, including the following:

- How to retrieve all columns and rows from a table
- How to retrieve specific columns of all rows in a table
- How to "describe" the structure of a table
- How to specify an alias for a column in a query

EXERCISE 4.1: Retrieving All Columns and Rows from a Table

Using SQL*Plus, enter the following query to retrieve all columns and rows from the ORDERS table.

```
SELECT * FROM orders;
```

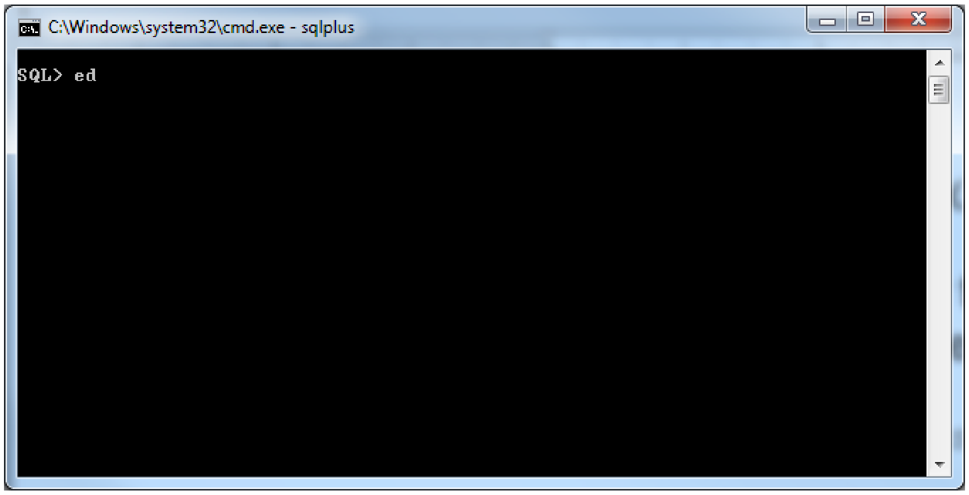
You have seen the results of the query above. For the sake of easy understanding, let us reprint the results as follows:

```
SQL> SELECT * FROM orders;

        ID       C_ID ORDERDATE SHIPDATE  PAIDDATE  S
---------- ---------- --------- --------- --------- -
         1          1 18-JUN-99 18-JUN-99 30-JUN-99 F
         2          2 18-JUN-99                     B
         3          3 18-JUN-99 18-JUN-99 21-JUN-99 F
         4          4 19-JUN-99 21-JUN-99 21-JUN-99 F
         5          5 19-JUN-99 19-JUN-99 28-JUN-99 F
         6          6 19-JUN-99 19-JUN-99           F
         7          7 19-JUN-99                     B
         8          8 20-JUN-99 20-JUN-99 20-JUN-99 F
         9          9 21-JUN-99                     B
        10          2 21-JUN-99 22-JUN-99 22-JUN-99 F
        11          4 22-JUN-99 22-JUN-99           F
        12          7 22-JUN-99 23-JUN-99 30-JUN-99 F
        13          4 22-JUN-99                     B
        14          1 23-JUN-99 25-JUN-99           F

14 rows selected.
```

The wildcard asterisk character (*) in the SELECT clause indicates that the query should retrieve all columns from the target table.

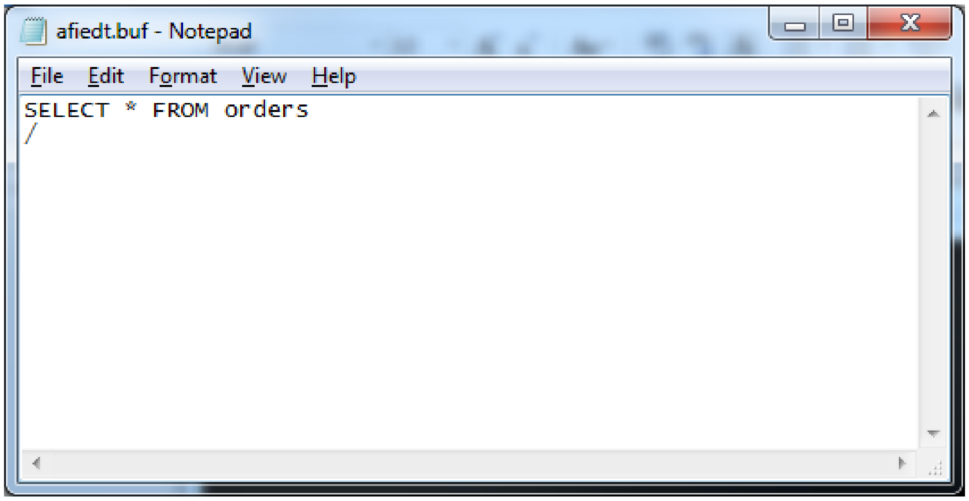
The above query is very simple, and thus it is easy to correct if we make a mistake in typing the command. It may be frustrating if we have to deal with more complex SQL query in the future. One helpful SQL*Plus feature is to invoke a text editor. The following shows you how to do it.

At SQL prompt, type “ed” —It will start your default text editor.



```
SQL> ed
```

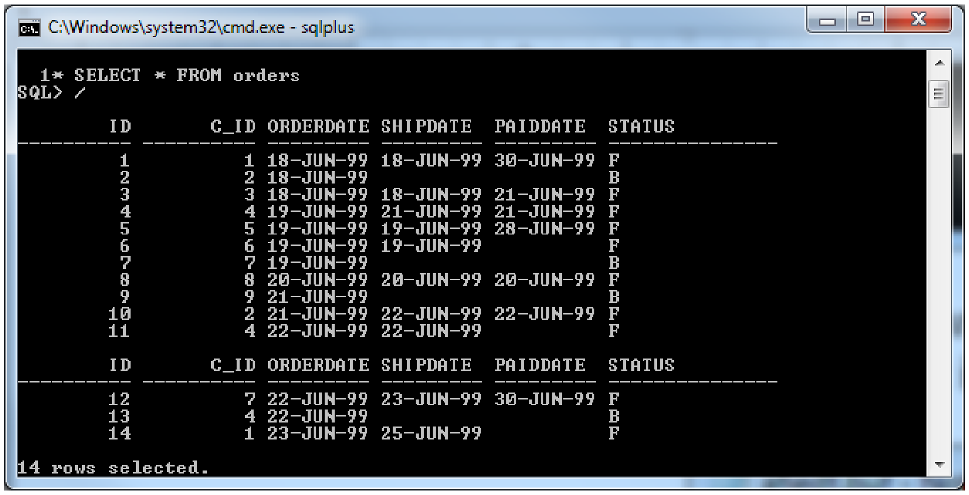
Then, you will see the following screen for your default text editor (in Windows, it is Notepad.)



```
SELECT * FROM orders
/
```

You may make whatever change you want, within your text editor. Save the changes, and exit. You will go back to SQL*Plus.

At the SQL> promot, you can type “/”. It will execute the revised SQL command for you.



```
  1* SELECT * FROM orders
SQL> /

        ID       C_ID ORDERDATE SHIPDATE  PAIDDATE  STATUS
---------- ---------- --------- --------- --------- ---------------
         1          1 18-JUN-99 18-JUN-99 30-JUN-99 F
         2          2 18-JUN-99                     B
         3          3 18-JUN-99 18-JUN-99 21-JUN-99 F
         4          4 19-JUN-99 21-JUN-99 21-JUN-99 F
         5          5 19-JUN-99 19-JUN-99 28-JUN-99 F
         6          6 19-JUN-99 19-JUN-99           F
         7          7 19-JUN-99                     B
         8          8 20-JUN-99 20-JUN-99 20-JUN-99 F
         9          9 21-JUN-99                     B
        10          2 21-JUN-99 22-JUN-99 22-JUN-99 F
        11          4 22-JUN-99 22-JUN-99           F

        ID       C_ID ORDERDATE SHIPDATE  PAIDDATE  STATUS
---------- ---------- --------- --------- --------- ---------------
        12          7 22-JUN-99 23-JUN-99 30-JUN-99 F
        13          4 22-JUN-99                     B
        14          1 23-JUN-99 25-JUN-99           F

14 rows selected.
```

I found the above extremely handy.

If you are tired of too much junk on your SQL*Plus screen, you may clean the screen by issue the following SQL command:

SQL>cl scr;

I use “cl scr” very often.

EXERCISE 4.2: Retrieving Specific Columns from a Table

To retrieve specific columns from all rows in a table, a query's SELECT clause must explicitly specify the name of each column to retrieve. For example, enter the following statement to retrieve just the ID and ORDERDATE columns for all rows in the ORDERS table.

```
SELECT id, orderdate FROM orders;
```

The result set is as follows:

```
SQL> SELECT id, orderdate FROM orders;

        ID ORDERDATE
---------- ---------
         1 18-JUN-99
         2 18-JUN-99
         3 18-JUN-99
         4 19-JUN-99
         5 19-JUN-99
         6 19-JUN-99
         7 19-JUN-99
         8 20-JUN-99
         9 21-JUN-99
        10 21-JUN-99
        11 22-JUN-99
        12 22-JUN-99
        13 22-JUN-99
        14 23-JUN-99

14 rows selected.
```

*EXERCISE 4.3: Using the SQL*Plus DESCRIBE Command*

If you are using SQL*Plus and you do not know the names of the columns in a table that you would like to query, use the special SQL*Plus command DESCRIBE to output the structure of the table. Enter the following DESCRIBE statement to display the column names of the ORDERS table (as well as additional information for each column).

```
DESCRIBE orders;
```

The results of the previous command should be similar to the following:

```
Name                                      Null?    Type
----------------------------------------- -------- ------
ID                                        NOT NULL
NUMBER(38)
C_ID                                      NOT NULL
NUMBER(38)
ORDERDATE                                 NOT NULL DATE
SHIPDATE                                           DATE
PAIDDATE                                           DATE
STATUS
```

EXERCISE 4.4: Specifying an Alias for a Column

To customize the names of the columns in a query's result set, you have the option of specifying an alias (an alternate name) for each column (or expression) in the query's SELECT clause. To rename a column or expression in the SELECT clause, just specify an alias after the list item.

NOTE

You must delimit the alias with double quotes if you want to specify the case, spacing, or include special characters in an alias.

To use a column alias, enter the following query, which specifies an alias for the ONHAND column of the PARTS table.

```
      SELECT id, onhand AS "IN STOCK"
FROM parts;
```

The result set is as follows:

```
SQL> SELECT id, onhand AS "IN STOCK"
  2  FROM parts;

        ID   IN STOCK
---------- ----------
         1        277
         2        143
         3       7631
         4       5903
         5        490
```

As this example demonstrates, you can precede a column alias with the optional keyword AS to make the alias specification more readable.

4.2.3 Building Expressions in a Query's SELECT Clause

In addition to simple column specifications, the SELECT clause of a query can also include expressions. An expression is a SQL construct that derives a character string, date, or numeric value. There are several different types of constructs that you can use to build expressions in a query's SELECT clause and return the resulting data in the query's result set, including operators, SQL functions, and decoded expressions. The next few sections provide a brief introduction to SELECT clause expressions.

EXERCISE 4.5: Building SELECT Clause Expressions with the Concatenation String Operator

An operator is a symbol that transforms a column value or combines it somehow with another column value or literal (an explicit value). For example, a simple way to build an expression in a SELECT clause is to use the concatenation operator—two solid vertical bars (||)—to concatenate (combine) character columns and/or string literals.

Enter the following query, which includes a simple SELECT clause expression. For each record in the CUSTOMERS table, the expression concatenates the LASTNAME field with a comma and a blank space (delimited by single quotes), and then the resulting string with the FIRSTNAME field. The expression also has a column alias to make its column heading in the result set more readable.

```
    SELECT lastname || ', ' || firstname AS name
FROM customers;
```

The result set is as follows:

```
SQL> SELECT lastname || ', ' || firstname AS name
  2  FROM customers;

NAME
--------------------------------------------------------
Ayers, Jack
Clay, Dorothy
Elias, Juan
Foss, Betty
Haagensen, Dave
Joy, Harold
Musial, Bill
Sams, Danielle
Schaub, Greg
Wiersbicki, Joseph

10 rows selected.
```

NOTE

The expressions on which an operator acts are called operands. The concatenation operator is an example of a binary operator—an operator that takes two operands and creates a new result. An operand that creates a value from a single operand is called a unary operator.

EXERCISE 4.6: Building SELECT Clause Expressions with Arithmetic Operators

Oracle's unary and binary arithmetic operators are listed in Table 4-1. An arithmetic operator accepts one or more numeric operands and produces a single numeric result.

Let's try a query that contains a SELECT clause expression that uses a binary arithmetic operator. Enter the following query, which determines how many of each part remains in inventory above the corresponding part's reorder threshold.

```
SELECT id, onhand, reorder, onhand - reorder AS threshold
FROM parts;
```

The result set is as follows:

```
SQL> SELECT id, onhand, reorder, onhand - reorder AS threshold FROM parts;

        ID     ONHAND REORDER                                   THRESHOLD
---------- ---------- ---------------------------------------- ----------
         1        277 50                                              227
         2        143 25                                              118
         3       7631 1000                                           6631
         4       5903 1000                                           4903
         5        490 200                                             290
```

Arithmetic Operator	**Description**
+x (unary)	Indicates that x is positive
-x (unary)	Indicates that x is negative
x II y (binary)	Concatenates x and y
x + y (binary)	Adds x and y
x - y (binary)	Subtracts y from x
x * y (binary)	Multiplies x by y
x / y (binary)	Divides x by y

TABLE 4-1. The Arithmetic Operators Supported by Oracle

EXERCISE 4.7: Building SELECT Clause Expressions with SQL Functions

You can also build an expression in a query's SELECT clause by using one or more of SQL's built-in functions. A function takes zero, one, or multiple arguments and returns a single value. There are two general types of SQL functions that you can use with queries: single-row functions and group functions.

A single-row (or scalar) function returns a value for every row that is part of a query's result set. Oracle supports many different categories of single-row SQL functions, including character, date, numeric, and conversion functions. For example, enter the following query, which uses the SQL functions UPPER and LOWER in SELECT clause expressions to display the company name in uppercase letters for each customer record and the last name of each customer record in lowercase letters.

```
SELECT UPPER(companyname) AS companyname,
LOWER(lastname) AS lastname
FROM customers;
```

The result set is as follows:

```
COMPANYNAME                    LASTNAME
------------------------------ ---------------
MCDONALD CO.                   joy
CAR AUDIO CENTER               musial
WISE TRUCKING                  sams
ROSE GARDEN INN                elias
FOSS PHOTOGRAPHY               foss
PAMPERED PETS                  schaub
KEY LOCKSMITH                  wiersbicki
PARK VIEW INSURANCE            ayers
KENSER CORP.                   clay
DAVE'S TREE SERVICE            haagensen

10 rows selected.
```

If your display is not as neat as above, you may need the following SQL commands and rerun the above exercise:

```
SQL> COLUMN COMPANYNAME FORMAT A30;

SQL>COLUMN LASTNAME FORMAT A15;
```

The above commands define the width of the columns for display within SQL* Plus. They do not change any data from the table. You may do the same with other columns if you see the needs.

Now enter the following query, which uses two numeric single-row SQL functions in SELECT clause expressions, SQRT and LN, to display the square root of 49 and the natural logarithm of 10, respectively.

```
SELECT SQRT(49), LN(10)
FROM DUAL;
```

The result set is as follows:

```
 SQRT(49)     LN(10)
---------- ----------
         7 2.30258509
```

NOTE

The previous query targets a special table called DUAL that is present in every Oracle database. DUAL is an empty table, consisting of one column *and one row, which you can use to satisfy the requirement for a table in a query's FROM clause. DUAL is useful for queries that just perform arithmetic or use 3 SQL function to return a value.*

A group (or aggregate) function returns an aggregate value for all rows that are part of a query's result set. For example, enter the following query, which uses the group SQL functions COUNT, MAX, MIN , and AVG to return the number of records in the PARTS table, as well as the maximum, minimum, and average UNITPRICE for records in the PARTS table, respectively.

```
SELECT COUNT(id) AS count,
MAX(unitprice) AS max_price,
MIN(unitprice) AS min_price,
AVG(unitprice) AS ave_price
FROM parts;
```

The result set is as follows:

```
     COUNT  MAX_PRICE  MIN_PRICE  AVE_PRICE
---------- ---------- ---------- ----------
         5       4895         99     1718.6
```

The example queries in this section have introduced just a few of the many SQL functions that are available with Oracle. See your Oracle documentation for a complete list of the SQL functions that you can use when building SQL statements.

EXERCISE 4.8: Working with Nulls in SELECT Clause Expressions

A null indicates the absence of a value. It does not mean zero, however. For example, the SHIPDATE field of a sales record in the ORDERS table will be null until the order ships, after which you can replace the null with an actual date value. When building SELECT clause expressions for queries, you must pay special attention to the possibility of nulls; otherwise you might obtain inaccurate or nonsense query results.

In most cases, an expression that includes a null evaluates to null. For example, if you input the following SQL command

```
SELECT id,fax FROM customers;
```

You will see the following result as follows:

```
        ID FAX
---------- ------------------------------
         1
         2 775-859-2121
         3 203-955-9532
         4 214-907-3188
         5 215-543-9800
         6 602-617-7321
         7 718-445-8799
         8
         9 916-672-8753
        10

10 rows selected.
```

To explicitly handle nulls in SELECT clause expressions, you can use the special scalar SQL function NVL to return a value of your choice when a column is null. For example, enter the following query that substitutes the string literal "UNKNOWN" when the FAX field of a record in the CUSTOMERS table is null.

```
SELECT id,NVL(fax,’UNKNOWN’)AS Fax
FROM customers;
```

The result set is as follows:

```
        ID FAX
---------- ------------------------------
         1 UNKNOWN
         2 775-859-2121
         3 203-955-9532
         4 214-907-3188
         5 215-543-9800
         6 602-617-7321
         7 718-445-8799
         8 UNKNOWN
         9 916-672-8753
        10 UNKNOWN

10 rows selected.
```

EXERCISE 4.9: Using Decoded SELECT Clause Expressions

A decoded expression is a special type of SELECT clause expression that you use to translate codes or symbols in a column. For example, enter the following query, which includes a decoded SELECT clause expression to convert the codes F and B of the STATUS column in the ORDERS table to the corresponding string literals FILLED and BACKORDERED.

```
SELECT id, DECODE (status,‘F','FILLED','B','BACKORDERED’, ‘OTHER’)
AS Status FROM orders;
```

The result set is as follows:

```
        ID STATUS
---------- -----------
         1 FILLED
         2 BACKORDERED
         3 FILLED
         4 FILLED
         5 FILLED
         6 FILLED
         7 BACKORDERED
         8 FILLED
         9 BACKORDERED
        10 FILLED
        11 FILLED
        12 FILLED
        13 BACKORDERED
        14 FILLED

14 rows selected.
```

If you see a line between some rows, you may find the following SQL*Plus command useful:

```
SQL>Set pagesize 60;
```

The above command defines the number of lines SQL*Plus displays before it breaks.

The above example demonstrates that to specify a decoded expression in a SELECT clause, you use the DECODE ( ) keyword and several parameters. The first parameter is the expression to be evaluated, which can be as simple as a column in a table. The second parameter is a comma-separated list of value pairs. For each possible value of the expression in the first parameter, specify a corresponding display value. The final parameter, which is optional, is a default display value for all expression values that are not specified in the list of value pairs. In the example here, the default display value is the string literal "OTHER."

4.2.4 Retrieving Specific Rows from Tables

So far, all of the example queries in this chapter show how to retrieve every row from a table. Optionally, a query can limit the result set to those rows in the target table that satisfy a WHERE clause condition. If the Boolean condition of the WHERE clause evaluates to TRUE for a particular row, Oracle includes the row in the query's result set. For example, the following query includes a WHERE clause condition that limits the result set to only those records in the ORDERS table that have a STATUS code equal to B.

```
SELECT id AS backorders, orderdate
FROM orders
WHERE status ='B';
```

The result set is as follows:

```
BACKORDERS ORDERDATE
---------- ---------
         2 18-JUN-99
         7 19-JUN-99
         9 21-JUN-99
        13 22-JUN-99
```

NOTE

If you submit a query and omit a WHERE clause, Oracle selects all rows from the targets specified in the query's FROM clause.

The next three exercises explain how to build WHERE clause conditions with relational operators, subqueries, and logical operators.

EXERCISE 4.10: Building WHERE Clause Conditions with Relational Operators

Oracle supports several relational operators that you can use to build WHERE clause conditions. A relational (or comparison) operator compares two operands to determine whether they are equal, or if one operand is greater than the other, A condition that uses a relational operator produces a Boolean value of TRUE or FALSE; however, if a null is involved in the condition, some relational operators produce an UNKNOWN result. Table 4-2 lists the relational operators that Oracle supports.

Relational Operator	Description
x = y	Determines if x equals y*
x ! = y x ^ = y x <>y	Determines if x is not equal to y*
x <y	Determines if x is less than y.*
x >= y	Determines if x is greater than or equal to y*
x <= y	Determines if x is less than or equal to y.
x lN (*list* | *subquery*)	Determines if the operand x is present in the list of values specified or returned by a subquery.
x NOT IN (*list* | *subquery*)	Determines if operand x is not present in the list of values specified by a subquery. The evaluation is FALSE when a list member is null.

x {=|!=|>|<|>=|<=}
{ ANY l SOME}
(list l subquery)
	Compares the operand x to the list of values specified or returned by a subquery. The evaluation is FALSE when a subquery does not return any rows. The keywords ANY and SOME are equivalent.

x	Compares the operand x to the list of values

{=|!=|>|<|>=|<=} ALL (list l subquery)	specified or returned by a subquery. The evaluation is TRUE when the subquery does not return any rows.
x BETWEEN y AND z	Determines if x falls within the exclusive range of y and z.
x NOT BETWEEN y AND z	Determines if x does not fall within the exclusive range of y and z.
EXISTS (subquery)	Determines if the subquery returns rows. The evaluation is TRUE when the subquery returns one or more rows, and FALSE when the subquery does not return any rows.
x LIKE y [ESCAPE z)	Determines if x matches the pattern y. y can include the wildcard characters % and _. % matches any string of zero or more characters, except null. _ matches any single character. Use the optional ESCAPE parameter only when you want Oracle to interpret literally a wildcard character as the escape character.
x NOT LIKE y [ESCAPE z]	Determines if x does not match the pattern y. y can include the wildcard characters % and _, and you can include the optional ESCAPE parameter (explained above).
x IS NULL	Determines if x is null.
x > y	Determines if x is greater than y.*
x IS NOT NULL	Determines if x is not null.

** If y is subquery, it must return a single row.*

TABLE 4-2. The Relational Operators Supported by Oracle

Let's try a couple of queries that use relational operators to build WHERE clause conditions. First, enter the following query to display the ID and ORDERDATE of all records in the ORDERS tables that have an ORDERDATE greater than June 21, 1999.

```
SELECT id, orderdate
FROM orders
WHERE orderdate > '21-JUN-99';
```

The result set is as follows:

```
      ID ORDERDATE
---------- ---------
      11 22-JUN-99
      12 22-JUN-99
      13 22-JUN-99
      14 23-JUN-99
```

Next, enter the following query to list the ID and LASTNAME of all customer records that do not have a FAX number. Notice that the WHERE clause condition uses the special IS NULL relational operator to test for the presence of a null in each record.

```
SELECT id, lastname FROM customers
WHERE fax IS NULL;
```

The result set is as follows:

```
        ID LASTNAME
---------- ---------------
         1 Joy
         8 Ayers
        10 Haagensen
```

EXERCISE 4.11: Building WHERE Clause Conditions with Subqueries

Some queries need to retrieve the answers based on multiple-part questions. For example, how would you build a query that reports the IDs of all orders placed by customers of a particular sales representative? This is really a two-part question, because the ORDERS table does not contain a field to record the sales representative that made the sale. However, the ORDERS table does include a field for the customer that made the purchase, and the CUSTOMERS table uses the S_ID column to indicate the sales representative for each customer record. Therefore, you can answer the original question by first building a list of customers that correspond to the sales representative, and then using the list to retrieve the IDs of the orders that correspond to just those customers.

To answer a multiple-part question in SQL with just one query, the query can use a WHERE clause condition with a subquery. A subquery is a technique that you can use to build an operand for the right side of a WHERE clause condition that uses a relational operator. When a WHERE clause condition uses a subquery, Oracle evaluates the subquery (or inner, nested query) first to return a value or list of values to the relational operator before evaluating the main (or outer) query.

Let's try a query that uses a subquery to answer the question asked at the beginning of this section. Enter the following query, which uses a WHERE clause condition with the relational operator IN and a subquery to report the ID and ORDERDATE of all orders placed by customers of sales representative number 2.

```
SELECT id, orderdate
FROM orders
WHERE c_id IN ( SELECT id FROM customers WHERE s_id = 2 );
```

The result set is as follows:

```
        ID ORDERDATE
---------- ---------
         4 19-JUN-99
        11 22-JUN-99
        13 22-JUN-99
```

Now consider what you would do if you needed to answer the previous question, but you did not know the ID of the sales representative with the last name "Jonah." In this case, you would need to answer a three-part question with a single query. First, you would need to get the ID of the sales representative named Jonah, then get a list of Jonah's customers, and then list the ID and ORDERDATE that correspond to those customers only. To accomplish this feat with a single query, the subquery of the main query must use a subquery itself. Enter the following query to try this out for yourself.

```
SELECT id, orderdate
FROM orders
WHERE c_id IN ( SELECT id FROM customers WHERE s_id = ( SELECT
id FROM salesreps WHERE lastname = 'Jonah') );
```

The result set is as follows:

```
        ID ORDERDATE
---------- ---------
         4 19-JUN-99
        11 22-JUN-99
        13 22-JUN-99
```

Notice that the innermost subquery uses the equals relational operator (=) and a subquery to determine the ID of the sales representative with the last name "Jonah." Oracle supports the use of a subquery with many relational operators—see Table 4-2 for a complete list of all relational operators that support subqueries, and their corresponding descriptions.

EXERCISE 4.12: Building Composite WHERE Clause Conditions with Logical Operators

Oracle also supports the use of the logical operators AND and OR to build a composite WHERE clause condition—a condition that includes two or more conditions.

- The AND operator combines the results of two conditions. The entire condition evaluates to TRUE only when both conditions on either side of the AND operator are TRUE. It evaluates to FALSE when either or both conditions are FALSE, and it evaluates to UNKNOWN when both conditions are null, or when either condition is TRUE and the other is null.

- The OR operator combines the results of two conditions. The entire condition evaluates to FALSE only when both conditions on either side of the OR operator are FALSE. It evaluates to TRUE when either or both conditions are TRUE, and it evaluates to UNKNOWN when both conditions are null, or when either condition is FALSE and the other is null.

Enter the following query, which uses the AND logical operator to build a composite WHERE clause condition that reports the ID and ORDERDATE of records in the ORDERS table that were ordered on or after June 1, 1999, and have a STATUS equal to B (on back order).

```
SELECT id, orderdate
FROM orders
WHERE orderdate >= ‘1-JUN-99’ AND status = ‘B’;
```

The result set is as follows:

```
        ID ORDERDATE
---------- ---------
         2 18-JUN-99
         7 19-JUN-99
         9 21-JUN-99
        13 22-JUN-99
```

NOTE

Oracle also supports the logical operator NOT so that you can build a query with a WHERE clause condition that tests for records that contradict a condition.

4.2.5 Grouping and Sorting Data within a Query's Result Set

The previous exercises in this chapter teach you how to include specific columns and rows in a query's result set. The next two sections explain how to format the output of a query's result set by grouping and sorting the records.

EXERCISE 4.13: Grouping Records in a Query's Result Set

Many queries need to answer questions based on aggregates or summaries of information rather than the details of individual records in a table. To group data in a query's result set, the query must include a GROUP BY clause. The list of columns in the GROUP BY clause specify how to aggregate the rows in the result set. A GROUP BY clause can list any column name in the table that appears in the query's FROM clause; however, a GROUP BY clause cannot reference aliases defined in the SELECT clause.

When you build the SELECT clause of a query that includes a GROUP BY clause, you can include an expression that uses a group function, including the functions AVG, COUNT, MAX, MIN, STDDEV, SUM, and VARIANCE. You can also include an unaggregated column or expression in the SELECT clause, but the same column or expression must be one of the columns or expressions in the GROUP BY clause.

Let's try grouping some data in a query's result set by entering the following query, which uses the group SQL function COUNT and a GROUP BY clause to display the number of orders placed by each customer.

```
SELECT c_id AS customer,
COUNT(id) AS orders_placed
FROM orders
GROUP BY c_id;
```

The result set is as follows:

```
  CUSTOMER ORDERS_PLACED
---------- -------------
         1             2
         6             1
         2             2
         4             3
         5             1
         8             1
         3             1
         7             2
         9             1

9 rows selected.
```

To eliminate selected groups from a query's result set, you can add a HAVING clause to the query's GROUP BY clause. Much like a WHERE clause condition, Oracle includes only those groups in a query's result set that evaluate to TRUE for the HAVING clause condition. For example, enter the following query, which displays the ID and number of orders placed by customers that have placed more than one order.

```
SELECT c_id AS customer,
COUNT(id) AS orders_placed
FROM orders
GROUP BY c_id
HAVING COUNT(id) > 1;
```

The result set is as follows:

```
  CUSTOMER ORDERS_PLACED
---------- -------------
         1             2
         2             2
         4             3
         7             2
```

NOTE

A query can use both a WHERE clause and the HAVING clause of a GROUP BY clause to eliminate data from the query's result set. Before forming groups, Oracle first removes all rows from the query's result set that do not satisfy the condition of the WHERE clause. Next, Oracle uses the expressions in the GROUP BY clause to form summary groups. Finally, Oracle removes the groups in the result set that do not satisfy the condition of the HAVING clause.

EXERCISE 4.14: Ordering Records in a Query's Result Set

A query can sort the rows in its result set in ascending or descending order by including an ORDER BY clause. You can sort rows by any number of columns and expressions that are in the query's SELECT clause, or by any column in the target table even if it is not included in the SELECT clause. In the ORDER BY clause, specify a comma-separated list of the columns and expressions to sort on, either by name, by their position in the SELECT clause, or by their alias in the SELECT list, and indicate whether you want ascending (ASC) or descending (DESC] order for each item. The default sort order for all columns and expressions in an ORDER BY clause is ascending order.

Enter the following query, which displays each customer record's ID, LASTNAME, and ZIPCODE, sorted in ascending order by the record's ZIPCODE.

```
SELECT id, lastname, zipcode
FROM customers
ORDER BY zipcode ASC;
```

The result set is as follows:

```
        ID LASTNAME        ZIPCODE
---------- --------------- -------------------------
         3 Sams            06103
         7 Wiersbicki      11220
         5 Foss            19144
         1 Joy             21209
        10 Haagensen       44124
         8 Ayers           66604
         4 Elias           75252
         6 Schaub          85023
         2 Musial          89501
         9 Clay            95821

10 rows selected.
```

NOTE

When you order query's result set by an expression that returns nulls, Oracle places the rows with nulls last in the result set when you order in ascending order; when you order in descending order, Oracle places the rows with nulls first in the result set.

4.2.6 Joining Data in Related Tables

The previous examples in this chapter are all queries that target data from only one table. When you work with a relational database especially a large database, you will face the task to pull information from multiple tables. That is where fun starts.

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

When you want to join information from N number of tables, include a comma-separated list of the tables to join in the query's FROM clause. The query's WHERE clause should have (N-1) join conditions that explain how to relate the data in the tables. The join query's SELECT clause can contain columns and expressions that refer to some or all of the columns in the target tables; however, when two columns in different tables have identical names, you must qualify references to these columns throughout the query with a table name to avoid ambiguity. The following two exercises contain examples of these syntax rules and explain two specific types of join queries: inner joins and outer joins.

EXERCISE 4.15: Building an Inner Join of Two Tables

An inner join (sometimes also known as simple join) combines the rows of two related tables based on a common column (or combination of common columns). The result set of an inner join does not include rows in either table that do not have a match in the other table.

To specify an inner-join operation, use a WHERE clause condition that relates the common columns in each table as operands on either side of the equality operator. For example, enter the following query that performs an inner join of the CUSTOMERS and ORDERS table.

```
SELECT c.id AS customer_id,
o.id AS order_id
FROM customers c,orders o
WHERE c.id = o.c_id
ORDER BY c.id;
```

The result set is as follows:

```
CUSTOMER_ID   ORDER_ID
----------- ----------
          1          1
          1         14
          2          2
          2         10
          3          3
          4          4
          4         11
          4         13
          5          5
          6          6
          7          7
          7         12
          8          8
          9          9

14 rows selected.
```

This example of an inner join retrieves a cross-product of all rows in the CUSTOMERS table that have placed orders. Notice that the result set of this query does not include any row for the customer with an ID of 10, because that customer has not placed any orders.

The previous example also shows how to declare aliases for tables in a query's FROM clause, and to use the aliases to qualify column names in other clauses of the query. For example, alias “c” was used for table “customers” and “o” for “orders.” The qualification of a column name with its table name is necessary when the two tables in the join have columns with the same name.

Many students new to Oracle may attempt to join tables without fully understanding the joining condition. For the above example, when we use the joining condition (WHERE c.id = o.c_id), we have to understand that c.id in “customers” table, and o.c_id in “orders” table are meant to be the same physical thing (customer id number). If we join tables with incorrect condition, sometimes, Oracle simply gives you results without any waring if the information is correct or not. Beware.

NOTE

When a join uses the equality comparison operator in the join condition, the query is also commonly referred to as an equi join.

For curiosity, you may want to experiment the above query by deleting the “`WHERE c.id = o.c_id`” clause. Oracle will generate so-called

Cartesian product. It combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. Actually, it is something we want to avoid in coding since it consumes resources.

Warning: Oracle 11g supports "natural join" function. Please note that "natural join" only applies when two tables have exactly the same column and column name. Otherwise, it will produce wrong output without telling you that you did it wrong. For example, the above "customers" table and "orders" table, both have a column named "id." But, the "id" in "customers" table is meant to be customer id, whereas "id" in "orders" table is meant to be order id. If we use "natural join," we will see wrong results. Please test it.

The easiest way to edit your code (including deleting a line) is to issue the following SQL* Plus command:

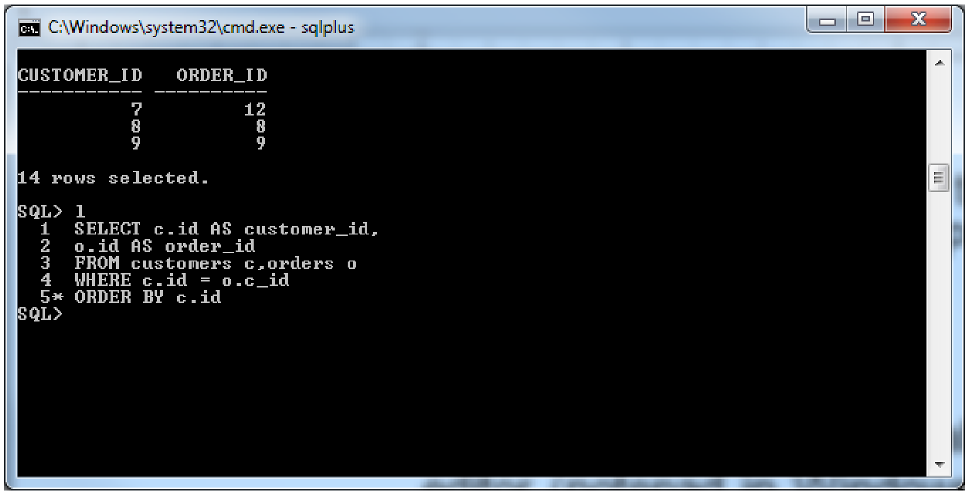
SQL> ed

You will then be able to edit your code, save it from your default text editor (notepad in Windows or vi in UNIX) and exit back to SQL* Plus. After that, you may execute the newly edited code by the following command:

SQL>/

To see your code in the SQL* Plus buffer, you may use:

SQL> L



```
CUSTOMER_ID   ORDER_ID
----------- ----------
          7         12
          8          8
          9          9

14 rows selected.

SQL> l
  1  SELECT c.id AS customer_id,
  2  o.id AS order_id
  3  FROM customers c,orders o
  4  WHERE c.id = o.c_id
  5* ORDER BY c.id
SQL>
```

EXERCISE 4.16: Building an Outer Join of Two Tables

Similar to an inner join, an outer join also combines the rows of two related tables based on a common column (or combination of common columns). However, the result set of an outer join includes rows from one of the tables even when there are not any matching rows in the other table.

The WHERE clause condition of an outer join is similar to that of an inner, equi join with one exception—you must place the outer-join operator, a plus sign delimited by parentheses (+), after one of the columns in the join condition. Specifically, the outer-join operator follows the column in the join condition that corresponds to the table for which you expect no matching rows. For example, enter the following query which performs an outer join of the CUSTOMERS and ORDERS tables, and includes all customers even when a customer does not have any matching orders.

```
SELECT c.id AS customer_id, o.id AS order_id
FROM customers c, orders o
WHERE c.id = o.c_id(+)
ORDER BY c.id;
```

The result set is as follows:

```
CUSTOMER_ID   ORDER_ID
----------- ----------
          1          1
          1         14
          2          2
          2         10
          3          3
          4          4
          4         11
          4         13
          5          5
          6          6
          7          7
          7         12
          8          8
          9          9
         10

15 rows selected.
```

Notice that the result set includes a row for the record in the CUSTOMERS table with an ID of 10, even though the customer has not placed any orders. Oracle returns nulls for all columns that have no match in an outer join's result set.

Oracle also refers the above outer join as **Left Outer Join** since it returns every row from the left table (CUSTOMERS). In Oracle 10g and 11g, Oracle offers a new way of doing the left outer join. You may enter the following code in SQL* Plus:

```
SELECT c.id AS customer_id, o.id AS order_id
FROM customers c LEFT OUTER JOIN orders o
ON c.id = o.c_id
ORDER BY c.id;
```

You will have the same results as the previous example. Please note the use of "`LEFT OUTER JOIN`" and "`ON`" in the FROM clause, and the "WHERE" clause is no longer needed in the new methods.

At this writing, Oracle recommends the use of the new method while continuing support the previous approach. I personally think it is a matter of preference. It is your choice.

As a developer, I would highly recommend that you test your join query very carefully before you put it in your program.

4.3 Inserting, Updating, and Deleting Rows in Tables

All of the previous examples in this chapter show you how to retrieve data from tables in an Oracle database using various queries (SELECT commands). Now let's take a look at how to input, modify, and delete table data using the SQL commands INSERT, UPDATE, and DELETE.

EXERCISE 4.17: Inserting New Rows into a Table

Before you start the exercise, it is a good idea to see what data you have in "PARTS" table, by using the following statement:

```
SELECT * FROM parts;
```

The following SQL* Plus commands may be helpful if you like to see the displayed data in a more organized way.

```
SQL> COLUMN DESCRIPTION FORMAT A15;

SQL> COLUMN REORDER FORMAT A8;
```

To insert a new row into a table, you use the SQL command INSERT. For example, enter the following statement, which inserts a new part into the PARTS table.

```
INSERT INTO
parts(id,description,unitprice,onhand,reorder)
VALUES(6,'Mouse',49,1200,500);
```

This example statement demonstrates how to use the most common clauses and parameters of the INSERT command.

- Use the INTO parameter to specify the target table.
- To insert a single row into the target table, use a VALUES clause that contains a comma-separated list of values for various columns in the table.
- Optionally, a comma-separated list of column names specifies the columns that correspond to the values in the VALUES clause. The number of columns in the list of column names must match the values in the VALUES clause. When you omit a column list altogether, the VALUES clause must specify values for every column in the target table in the order that the columns appear in the table.

Now, you can use the above SELECT statement to see the inserted row in your “PARTS” table.

Personal note: If you copy and paste the above code into Notepad and run SQL*Plus, you may encounter an error. It is due to the fact that a single quote (‘) in Microsoft Word does not mean the same in Notepad. If you retype the code in Notepad, it will not complain.

EXERCISE 4.18: Updating Rows in a Table

To update column values in one or more rows of a table, use the SQL command UPDATE. For example, enter the following UPDATE statement, which updates the UNITPRICE value for a part in the PARTS table.

```
UPDATE parts
SET unitprice = 55,onhand = 1100
WHERE id = 6;
```

Notice that the SET clause of the example UPDATE statement updates the UNITPRICE and ONHAND values of a specific record in the PARTS table—the record identified by the condition of the WHERE clause, Be careful—when an UPDATE statement omits selection criteria (in other words, a WHERE clause), the UPDATE statement updates all rows in the target table.

EXERCISE 4.19: Deleting Rows from a Table

To delete one or more rows from a table, you use the SQL command DELETE. For example, enter the following DELETE statement, which deletes a specific record from the PARTS table.

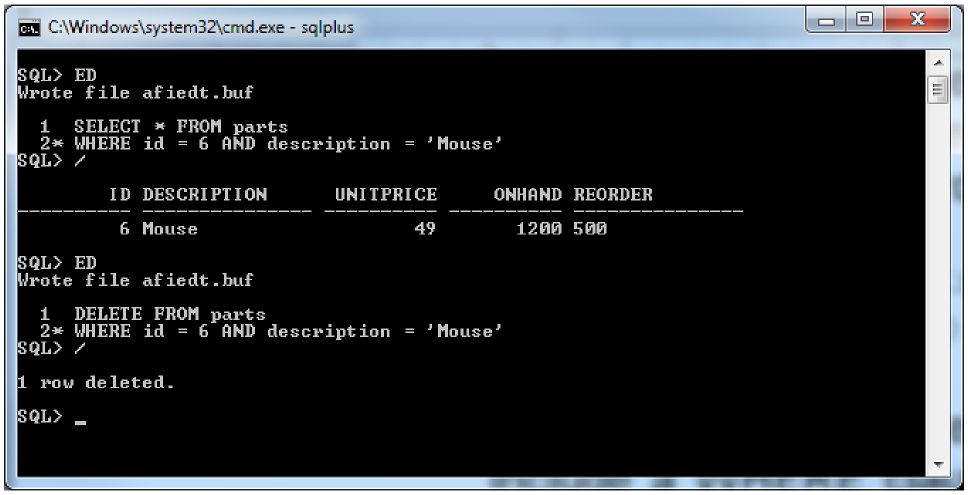
```
DELETE FROM parts
WHERE id = 6 AND description = 'Mouse';
```

As this example demonstrates, a DELETE statement should always include a WHERE clause with a condition that targets specific rows in a table, unless you want to delete all rows in the table.

As a database professional, you do not want to wipe out any data that you do not intend to delete, especially for a large database. To prevent this serious mistake, I always do a query first and then change the same query to delete statement. For example, before I issue the above delete command, I would do the following:

```
SELECT * FROM parts
WHERE id = 6 AND description = 'Mouse';
```

See the following screen shot:


C:\Windows\system32\cmd.exe - sqlplus

```
SQL> ED
Wrote file afiedt.buf

  1  SELECT * FROM parts
  2* WHERE id = 6 AND description = 'Mouse'
SQL> /

        ID DESCRIPTION      UNITPRICE     ONHAND REORDER
---------- --------------- ---------- ---------- ---------------
         6 Mouse                   49       1200 500

SQL> ED
Wrote file afiedt.buf

  1  DELETE FROM parts
  2* WHERE id = 6 AND description = 'Mouse'
SQL> /

1 row deleted.

SQL> _
```

The above may seem to be trivial. But, as a professional, it will pay off if you treat your data seriously.

4.4 Committing and Rolling Back Transactions (Transaction Control)

As you learned earlier in this chapter, a database transaction is a unit of work performed by one or more closely related SQL statements. The following hands-on exercise teaches you how to commit or roll back a transaction.

EXERCISE 4.20: Committing and Rolling Back Transactions

Enter the following series of related SQL statements, which together form a transaction that inserts a new order into the ORDERS table and two associated line items into the ITEMS table.

```
INSERT INTO orders
VALUES(15,3,’23-JUN-16’,’23-JUN-16’,NULL,‘F’);
```

```
INSERT INTO items
VALUES (15,1,4,1);

INSERT INTO items
VALUES (15,2,5,1);
```

To permanently commit the work of the transaction to the database, use the SQL command COMMIT (with or without the optional keyword WORK).

```
COMMIT WORK;
```

Alternatively, to undo or roll back the work of a transaction's SQL statements, use the SQL command ROLLBACK (with or without the optional keyword WORK).

```
ROLLBACK WORK;
```

NOTE

After you commit a transaction, Oracle automatically starts the next transaction. Therefore, you cannot roll back a transaction after you commit it.

A fundamental principle that you must clearly understand is that a transaction is a unit of work. That is, although a transaction might be made up of several SQL statements, they all commit or roll back as a single operation. For example, when an application commits a transaction, Oracle permanently records the changes made by all SQL statements in the transaction. If for some reason Oracle cannot commit the work of any statement in a transaction Oracle automatically rolls back the effects of all statements in the transaction.

When you start a database application and establish a connection to a database, Oracle implicitly starts a new transaction for your session. After you commit or roll back a transaction, Oracle again implicitly starts a new transaction for your session.

4.5 Transaction Design

The design of application transactions is very important, because a transaction's design can directly affect database integrity and the performance of applications. The following sections discuss several issues to consider when designing a database application's transactions.

4.5.1 Units of Work

Remember that a transaction is meant to encompass many closely related SQL statements that, together, perform a single unit of work. More specifically, a transaction should not encompass multiple units of work, nor should it encompass a partial unit of work. The following example demonstrates bad transaction design.

```
INSERT INTO Customers ... ;

INSERT INTO parts ... ,

INSERT INTO orders ... ;

INSERT INTO items ... ;

INSERT INTO items ... ;

COMMIT WORK ;
```

In this example, the bad transaction design encompasses three separate units of work.

1. The transaction inserts a new customer record.

2. The transaction inserts a new part record.

3. The transaction inserts the records for a new sales order.

Technically, each unit of work in the transaction has nothing to do with the others. When a transaction encompasses more than a single unit of work, Oracle must maintain internal system information on behalf of the transaction for a longer period of time. Quite possibly, this can detract from system performance, especially when many transactions burden Oracle with the same type of bad transaction design.

To contrast the previous type of bad transaction design, consider another example:

```
INSERT INTO orders ...;

COMMIT WORK ;

INSERT INTO items ... ;

COMMIT WORK ;

INSERT INTO items ... ;

COMMIT WORK ;
```

This example does the opposite of the previous example—there are three transactions to input the records for a single sales order. The overhead of many unnecessary small transaction commits can also detract from server performance. More important, partial transactions can risk the integrity of a database's data. For example, consider what would happen if you use the above transaction design to insert a new sales order, but before you can commit the insert of all line items for the new sales order, your session abnormally disconnects from the database server. At this point, the database contains a partial sales order, at least until you reconnect to finish the sales order. In the interim, a shipping transaction might look at the partial sales order and not realize that it is working with incomplete information. As a result, the shipping department might unknowingly send a partial product shipment to a customer and mark it as complete. The irate customer calls days later demanding to know why she didn't receive the other ordered products. When the shipping clerk looks at the order in the database, he sees the missing line items, but he cannot explain why the order did not contain the products and was marked as complete.

4.5.2 Read-Write Transactions

By default, when Oracle starts a new transaction for your session, the transaction is read-write. A read-write transaction can include any type of

SQL statement, including DML statements that query, insert, update, and delete table rows. To explicitly declare a transaction as a read-write transaction, you can begin the transaction with a SQL command SET TRANSACTION and the READ WRITE option.

```
SET TRANSACTION READ WRITE;
```

4.5.3 Read-Only Transactions

A read-only transaction includes queries only. In other words, a read-only transaction does not modify the database in any way. Certain reporting applications might want to explicitly declare a transaction as read-only with the READ ONLY option of the SET TRANSACTION command.

```
SET TRANSACTION READ ONLY;
```

When you declare an explicit read-only transaction, Oracle guarantees transaction-level read consistency for the transaction. This means that the result sets of all queries in the transaction reflect the database's data as it existed at the beginning of the transaction, even though other transactions modify and commit work to the database. Reporting applications commonly use an explicit read-only transaction to encompass several queries and produce a report with consistent data.

Class/Chapter Summary

This class/chapter has provided you with a broad overview of SQL and a tutorial for using the most common SQL commands.

- To accomplish work with Oracle, applications must use SQL commands.
- Use the SQL command SELECT to build queries that retrieve data from database tables. A query's FROM clause specifies the table(s) to target. A query's SELECT clause determines what columns to include in the query's result set—the SELECT clause can contain simple column names as well as expressions that use operators, literals, or SQL functions to derive data. A query's optional WHERE clause condition determines which rows to include in the query's result set. You can use the optional ORDER BY clause to sort the records in a query's result set. Use the optional GROUP BY clause to summarize or aggregate data in a query's result set.
- Use the SQL command INSERT to insert a row into a table. An INSERT statement's INTO parameter specifies the target table, and the VALUES clause contains a list of values for the various columns in the table. You can also include an optional list of column names before the VALUES clause to specify target columns in the table for the values in the VALUES clause.
- Use the SQL command UPDATE to update rows in a table. Use the SET clause to identify the column values to change, and a WHERE clause to target specific rows in the table that you want to update.
- Use the SQL command DELETE to delete rows from a table. A DELETE statement should always include a WHERE clause with a condition that targets specific rows in a table, unless you want to delete all rows in the table.
- Use the SQL commands COMMIT and ROLLBACK to control the outcome of your session's current transaction. A COMMIT statement permanently commits the changes made by all SQL statements in the transaction. A ROLLBACK statement undoes the effects of all SQL statements in the transaction. Oracle automatically starts a new transaction after you commit or roll back your current transaction.