


Oracle Database 11g: SQL Fundamentals I

PART

I

COPYRIGHTED MATERIAL





Chapter 1

Introducing SQL

ORACLE DATABASE 11g: SQL FUNDAMENTALS I EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Retrieving Data Using the SQL SELECT Statement**

- List the capabilities of SQL SELECT statements
- Execute a basic SELECT statement

✓ **Restricting and Sorting Data**

- Limit the rows that are retrieved by a query
- Sort the rows that are retrieved by a query
- Use ampersand substitution to restrict and sort output at runtime



Oracle 11g is a very powerful and feature-rich relational database management system (RDBMS). SQL has been adopted by most RDBMSs for the retrieval and management of data, schema creation, and access control. The American National Standards Institute (ANSI) has been refining standards for the SQL language for more than 20 years. Oracle, like many other companies, has taken the ANSI standard of SQL and extended it to include much additional functionality.

SQL is the basic language used to manipulate and retrieve data from the Oracle Database 11g. SQL is a nonprocedural language, meaning it does not have programmatic constructs such as loop structures. PL/SQL is Oracle's procedural extension of SQL, and SQLJ allows embedded SQL operations in Java code. The scope of the Oracle Database 11g SQL Fundamentals I test includes only SQL.

In this chapter, I will discuss Oracle SQL fundamentals such as the various types of SQL statements, introduce SQL*Plus and a few SQL*Plus commands, and discuss SELECT statements.

You will learn how to write basic SQL statements to retrieve data from tables. This will include coverage of SQL SELECT statements, which are used to query data from the database-storage structures, such as tables and views. You will also learn how to limit the information retrieved and to display the results in a specific order.



Exam objectives are subject to change at any time without prior notice and at Oracle's sole discretion. Please visit Oracle's Training and Certification website at http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?p_exam_id=120_051 for the most current exam objectives.

SQL Fundamentals

SQL is the standard language to query and modify data as well as manage databases. SQL is the common language used by programmers, database administrators, and users to access and manipulate data as well as to administer databases. To get started with SQL in this chapter, I will show how to use the sample HR schema supplied with the Oracle Database 11g.



When you install Oracle software, you can choose the Basic Installation option and select the Create Starter Database check box. This database will have the sample schemas used in this book. The password you specify will be applicable to the SYS and SYSTEM accounts. The account SYS is the Oracle dictionary owner, and SYSTEM is a database administrator (DBA) account. Initially, the sample schemas are locked. You need to log in to the database using SQL*Plus as the SYSTEM user and then unlock the account using the ALTER USER statement. To unlock the HR schema, use ALTER USER hr IDENTIFIED BY hrpassword ACCOUNT UNLOCK;. Now you can log in to the database using the hr user with the password hrpassword. Remember, the password is case sensitive.

For detailed information on installing Oracle 11g software and creating Oracle Database 11g, please refer to the Oracle Technology Network at www.oracle.com/technology/obe/11gr1_db/install/dbinst/windbinst2.htm.

To install the sample schemas in an existing Oracle Database 11g, please follow the instructions in the Oracle document "Oracle Database Sample Schemas 11g Release 1" at http://download.oracle.com/docs/cd/B28359_01/server.111/b28328/toc.htm.



Chapter 2 of the "Oracle Database Sample Schemas 11g Release 1" manual on the Oracle Technology Network will provide instructions on how to install the sample schemas using Database Configuration Assistant (DBCA) as well as running scripts. The same chapter also gives you steps to reinitialize the sample schema data.

SQL statements are like plain English but with specific syntax. SQL is a simple yet powerful language used to create, access, and manipulate data and structures in the database. SQL statements can be categorized as listed in Table 1.1.

TABLE 1.1 SQL Statement Categories

SQL Category	Description
Data Manipulation Language (DML)	Used to access, create, modify, or delete data in the existing structures of the database. DML statements include those to query information (SELECT), add new rows (INSERT), modify existing rows (UPDATE), delete existing rows (DELETE), perform a conditional update or insert operation (MERGE), see an execution plan of SQL (EXPLAIN PLAN), and lock a table to restrict access (LOCK TABLE). Including the SELECT statement in the DML group is debatable within the SQL community, since SELECT does not modify data.

TABLE 1.1 SQL Statement Categories (*continued*)

SQL Category	Description
Data Definition Language (DDL)	Used to define, alter, or drop database objects and their privileges. DDL statements include those to create, modify, drop, or rename objects (CREATE, ALTER, DROP, RENAME), remove all rows from a database object without dropping the structure (TRUNCATE), manage access privileges (GRANT, REVOKE), audit database use (AUDIT, NOAUDIT) and add a description about an object to the dictionary (COMMENT).
Transaction Control	Used to group a set of DML statements as a single transaction. Using these statements, you can save the changes (COMMIT) or discard the changes (ROLLBACK) made by DML statements. Also included in the transaction-control statements are statements to set a point or marker in the transaction for possible rollback (SAVEPOINT) and to define the properties for the transaction (SET TRANSACTION).
Session Control	Used to control the properties of a user session. (A session is the point from which you are connected to the database until you disconnect.) Session-control statements include those to control the session properties (ALTER SESSION) and to enable/disable roles (SET ROLE).
System Control	Used to manage the properties of the database. There is only one statement in this category (ALTER SYSTEM).

Table 1.1 provides an overview of all the statements that will be covered in this book. Do not worry if you do not understand certain terms, such as *role*, *session*, *privilege*, and so on. I will cover all the statements in the coming chapters with many examples. In this chapter, I will begin with writing simple statements to query the database (SELECT statements). But first I'll go over some fundamentals.

SQL Tools: SQL*Plus

The Oracle Database 11g software comes with two primary tools to manage data and administer databases using SQL. SQL*Plus is a character-based command-line utility. SQL Developer is a graphical tool that has the capability to browse, edit, and manage database objects as well as to execute the SQL statements. On Windows platforms, these tools are located under the Application Development subfolder in the Oracle 11g program group.

On Linux and Unix platforms, you can find these tools in the bin directory under the Oracle software installation (\$ORACLE_HOME/bin).

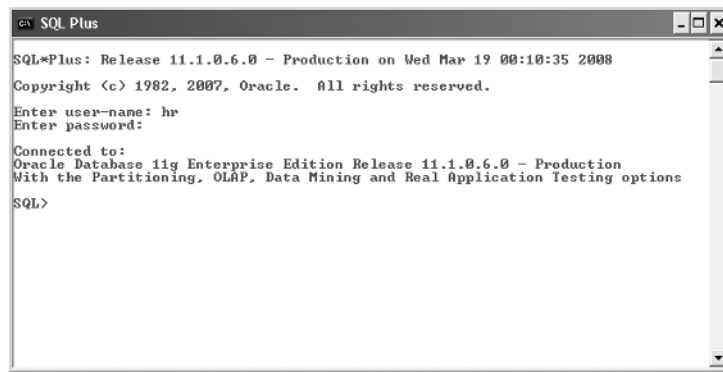
Since the test is on SQL and the tool used throughout the book for executing SQL is SQL*Plus, I will discuss some fundamentals of SQL*Plus in this section.

SQL*Plus, widely used by DBAs and developers to interact with the database, is a powerful tool from Oracle. Using SQL*Plus, you can execute all SQL statements and PL/SQL programs, format results from queries, and administer the database.

SQL*Plus is packaged with the Oracle software and can be installed using the client software installation routine on any machine. This tool is automatically installed when you install the server software.

On Unix/Linux platforms, you can invoke SQL*Plus using the `sqlplus` executable found in the `$ORACLE_HOME/bin` directory. On Windows and Unix/Linux platforms, when you start SQL*Plus, you will be prompted for a username and password, as shown in Figure 1.1.

FIGURE 1.1 SQL*Plus screen



Once you are in SQL*Plus, you can connect to another database or change your connection by using the `CONNECT` command, with this syntax:

```
CONNECT <username>/<password>@<connectstring>
```

The slash separates the username and password. The connect string following `@` is the database alias name. If you omit the password, you will be prompted to enter it. If you omit the connect string, SQL*Plus tries to connect you to the local database defined in the `ORACLE_SID` variable.

You can invoke and connect to SQL*Plus using the `sqlplus` command, with this syntax:

```
sqlplus <username>/<password>@<connectstring>
```

If you invoke the tool with just `sqlplus`, you will be prompted for a username and password. If you invoke SQL*Plus with a username, you will be prompted for a password.

Once you are connected to SQL*Plus, you get the `SQL>` prompt. This is the default prompt, which can be changed using the `SET SQLPROMPT` command. Type the command you want to

execute at this prompt. With SQL*Plus, you can enter, edit, and execute SQL statements; perform database administration; and execute statements interactively by accepting user input. You can also format query results and perform calculations.



`sqlplus -help` displays a help screen to show the various options available with starting SQL*Plus.

To exit from SQL*Plus, use the EXIT command. On platforms where a return code is used, you can provide a return code while exiting. You can also use the QUIT command to complete the session. EXIT and QUIT are synonymous.

Entering SQL Statements

A SQL statement can spread across multiple lines, and the commands are case insensitive. The previously executed SQL statement will always be available in the *SQL buffer*. The buffer can be edited or saved to a file. You can terminate a SQL statement in any of the following ways:

- End with a semicolon (;): The statement is completed and executed.
- Enter a slash (/) on a new line by itself: The statement in the buffer is executed.
- Enter a blank line: The statement is saved in the buffer.

You can use the RUN command instead of a slash to execute a statement in the buffer. The SQL prompt returns when the statement has completed execution. You can enter your next command at the prompt.



Only SQL statements and PL/SQL blocks are stored in the SQL buffer; SQL*Plus commands are not stored in the buffer.

Entering SQL*Plus Commands

SQL*Plus has its own commands to perform specific tasks on the database, as well as to format the query results. Unlike SQL statements, which are terminated with a semicolon or a blank line, SQL*Plus commands are entered on a single line. Pressing Enter executes the SQL*Plus command.

If you want to continue a SQL*Plus command onto the next line, you must end the current line with a hyphen (-), which indicates command continuation. This is in contrast to SQL statements, which can be continued to the next line without a continuation operator. For example, the following SQL statement gives an error, because SQL*Plus treats the hyphen operator (-) as a continuation character:

```
SQL> SELECT 800 -
> 400 FROM dual;
```



```
SELECT 800 400 FROM dual
      *
```

ERROR at line 1:

ORA-00923: FROM keyword not found where expected

SQL>

You need to put the hyphen in the next line for the query to succeed:

```
SQL> SELECT 800
```

```
      2 - 400 FROM dual;
```

```
      800-400
```

```
-----
```

```
      400
```

SQL>

Getting Information with the DESCRIBE Command

You can use the DESCRIBE command to get information about the database objects. Using DESCRIBE on a table or view shows the columns, its datatypes, and whether each column can be NULL. Using DESCRIBE on a stored program such as procedure or function shows the parameters that need to be passed in/out, their datatype, and whether there is a default value. You can abbreviate this command to the first four characters or more—DESC, DESCR, and DESCRIB are all valid.

If you're connected to the HR schema and need to see the tables and views in this schema, use the following query:

```
SQL> SELECT * FROM tab;
```

TNAME	TABTYPE	CLUSTERID
-----	-----	-----
COUNTRIES	TABLE	
DEPARTMENTS	TABLE	
EMPLOYEES	TABLE	
EMP_DETAILS_VIEW	VIEW	
JOBS	TABLE	
JOB_HISTORY	TABLE	
LOCATIONS	TABLE	
REGIONS	TABLE	

8 rows selected.

SQL>

Editing the SQL Buffer

The most recent SQL statement executed or entered is stored in the SQL buffer of SQL*Plus. You can run the command in this buffer again by simply typing a slash or using the RUN command.

SQL*Plus provides a set of commands to edit the buffer. Suppose you want to add another column or add an ORDER BY condition to the statement in the buffer. You do not need to type the entire SQL statement again. Instead, just edit the existing statement in the buffer.

One way to edit the SQL*Plus buffer is to use the EDIT command to write the buffer to an operating-system file named `afiedt.buf` (this is the default filename, which can be changed) and then use a system editor to make changes.



You can use your favorite text editor by defining it in SQL*Plus. For example, to make Notepad your favorite editor, just issue the command `DEFINE _EDITOR = NOTEPAD`. You need to provide the entire path if the program is not available in the search path.

Another way to edit the buffer is to use the SQL*Plus editing commands. You can make changes, delete lines, add text, and list the buffer contents using the commands described in the following sections. Most editing commands operate on the current line. You can change the current line simply by typing the line number. All commands can be abbreviated except DEL (which is already abbreviated).

LIST

The LIST command lists the contents of the buffer. The asterisk indicates the current line. The abbreviated command for LIST is L.

```
SQL> L
  1  SELECT empno,  ename
  2*  FROM emp
SQL> LIST LAST
  2*  FROM emp
SQL>
```

The command LIST *m n* displays lines from *m* through *n*. If you substitute * for *m* or *n*, it implies the current line. The command LIST LAST displays the last line.

APPEND

The APPEND *text* command adds text to the end of line. The abbreviated command is A.

```
SQL> A  WHERE empno <> 7926
  2*  FROM emp WHERE empno <> 7926
SQL>
```

CHANGE

The `CHANGE /old/new` command changes an old entry to a new entry. The abbreviated command is `C`. If you omit *new*, *old* will be deleted.

```
SQL> C /<>/=
      2* FROM emp WHERE empno = 7926
SQL> C /7926
      2* FROM emp WHERE empno =
SQL>
```

INPUT

The `INPUT text` command adds a line of text. Its abbreviation is `I`. If *text* is omitted, you can add as many lines you want.

```
SQL> I
      3 7777 AND
      4 empno = 4354
      5
SQL> I ORDER BY 1
SQL> L
      1 SELECT empno, ename
      2 FROM emp WHERE empno =
      3 7777 AND
      4 empno = 4354
      5* ORDER BY 1
SQL>
```

DEL

The `DEL` command used alone or with `*` deletes the current line. The `DEL m n` command deletes lines from *m* through *n*. If you substitute `*` for *m* or *n*, it implies the current line. The command `DEL LAST` deletes the last line.

```
SQL> 3
      3* 7777 AND
SQL> DEL
SQL> L
      1 SELECT empno, ename
      2 FROM emp WHERE empno =
      3 empno = 4354
      4* ORDER BY 1
SQL> DEL 3 *
```

```
SQL> L
  1  SELECT empno, ename
  2* FROM emp WHERE empno =
SQL>
```

CLEAR BUFFER

The CLEAR BUFFER command (abbreviated CL BUFF) clears the buffer. This deletes all lines from the buffer.

```
SQL> L
  1  SELECT empno, ename
  2* FROM emp WHERE empno =
SQL> CL BUFF
buffer cleared
SQL> L
No lines in SQL buffer.
SQL>
```

Using Script Files

SQL*Plus provides commands to save the SQL buffer to a file, as well as to run SQL statements from a file. SQL statements saved in a file are called a *script file*.

You can work with script files as follows:

- To save the SQL buffer to an operating-system file, use the command `SAVE filename`. If you do not provide an extension, the saved file will have an extension of `.sql`.
- By default, the SAVE command will not overwrite an existing file. If you want to overwrite an existing file, you need to use the keyword REPLACE.
- To add the buffer to the end of an existing file, use the `SAVE filename APPEND` command.
- You can edit the saved file using the `EDIT filename` command.
- You can bring the contents of a *script file* to the SQL buffer using the `GET filename` command.
- If you want to run a script file, use the command `START filename`. You can also run a script file using `@filename`.
- An `@@filename` used inside a script file looks for the filename in the directory where the parent *script file* is saved and executes it.

Exercise 1.1 will familiarize you with the script file commands, as well as the other topics I have covered so far.

EXERCISE 1.1**Practicing SQL*Plus File Commands**

In this exercise, you will learn how to edit the SQL*Plus buffer using various buffer edit commands.

1. Enter the following SQL; the third line is a blank line so that the SQL is saved in the buffer:

```
SQL> SELECT employee_id, first_name, last_name
      2 FROM   employees
      3
SQL>
```

2. List the SQL buffer:

```
SQL> L
      1 SELECT employee_id, first_name, last_name
      2* FROM   employees
SQL>
```

3. Save the buffer to a file named myfile; the default extension will be .sql:

```
SQL> SAVE myfile
Created file MYFILE.sql
SQL>
```

4. Choose to edit the file:

```
SQL> EDIT myfile
SQL>
```

5. Add WHERE EMPLOYEE_ID = 106 as the third line to the SQL statement.

6. List the buffer:

```
SQL> LIST
      1 SELECT employee_id, first_name, last_name
      2* FROM   employees
SQL>
```

The buffer listed is still the old buffer. The edited changes are not reflected because you edited the file MYFILE, which is not yet loaded to the buffer.

7. Bring the file contents to the buffer:

```
SQL> GET myfile
      1 SELECT employee_id, first_name, last_name
```

EXERCISE 1.1 (continued)

```

2 FROM employees
3* WHERE employee_id = 106
SQL>

```

8. List the buffer to verify its contents:

```

SQL> LI
1 SELECT employee_id, first_name, last_name
2 FROM employees
3* WHERE employee_id = 106
SQL>

```

9. Change the employee number from 106 to 110:

```

SQL> C/106/110
3* WHERE employee_id = 110
SQL>

```

10. Save the buffer again to the same file:

```

SQL> SAVE myfile
SP2-0540: File "MYFILE.sql" already exists.
Use "SAVE filename[.ext] REPLACE".
SQL>

```

An error is returned, because SAVE will not overwrite the file by default.

11. Save the file using the REPLACE keyword:

```

SQL> SAVE myfile REPLACE
Wrote file MYFILE.sql
SQL>

```

12. Execute the file:

```

SQL> START myfile

EMPLOYEE_ID FIRST_NAME          LAST_NAME
-----
          110 John              Chen
SQL>

```

13. Change the employee number from 110 to 106, and append this SQL to the file; then execute it using @:

```

SQL> C/110/106
3* WHERE employee_id = 106

```

EXERCISE 1.1 (continued)

```

SQL> SAVE myfile APPEND
Appended file to MYFILE.sql
SQL> @MYFILE
EMPLOYEE_ID FIRST_NAME          LAST_NAME
-----
          110 John              Chen

EMPLOYEE_ID FIRST_NAME          LAST_NAME
-----
          106 Valli             Pataballa
SQL>

```

Saving Query Results to a File

You can use the `SPPOOL filename` command to save the query results to a file. By default, the `SPPOOL` command creates an `.lst` file extension. `SPPOOL` overwrites an existing file by default. If you include the `APPEND` option as in `SPPOOL filename APPEND`, the results are added to an existing file. A new file will be created if the file does not exist already.

`SPPOOL OFF` stops writing the output to the file. `SPPOOL OUT` stops the writing of output and sends the output file to the printer.

Adding Comments to a Script File

Having comments in the script file improves the readability and understandability of the code. You can enter comments in SQL*Plus using the `REMARKS` (abbreviated `REM`) command. Lines in the script file beginning with the keyword `REM` are comments and are not executed. You can also enter a comment between `/*` and `*/`. Comments can also be entered following `--` (double hyphen), all characters following `--` in the line are treated as comment by Oracle.

While executing a script file with comments, the remarks entered using the `REMARKS` command are not displayed on the screen, but the comments within `/*` and `*/` are displayed on the screen with the prefix `DOC>` when there is more than one line between `/*` and `*/`. You can turn this off by using `SET DOCUMENT OFF`.

This section provided an overview of SQL*Plus, the tool you will be using to enter and execute SQL statements in Oracle Database 11g. In the next sections, I will discuss some of the Oracle 11g SQL fundamentals before showing you how to write your first SQL query (a `SELECT` statement).

Oracle Datatypes

The basic structure of data storage in the Oracle Database 11g is a table. A table can be considered as a spreadsheet with columns and rows. Data is stored in the table as rows. Each column in the table has storage characteristics such as the type of data contained in

the column. Oracle has several built-in datatypes to store different kinds of data. In this section, I will go over the built-in datatypes available in Oracle 11g. Detailed discussion on datatypes as well as creating and maintaining tables are discussed in Chapter 6, “Creating Tables and Constraints.”

When you create a table to store data in the database, you need to specify a datatype for all the columns you define in the table. Oracle has many datatypes to suit application requirements. Oracle 11g also supports ANSI and DB2 datatypes. The Oracle built-in datatypes can be broadly classified as shown in Table 1.2.

TABLE 1.2 Oracle Built-in Datatypes

Category	Datatypes
Character	CHAR, NCHAR, VARCHAR2, NVARCHAR2
Number	NUMBER, FLOAT, BINARY_FLOAT, BINARY_DOUBLE
Long and raw	LONG, LONG RAW, RAW
Date and time	DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND
Large object	CLOB, NCLOB, BLOB, BFILE
Row ID	ROWID, UROWID

In the following sections, I will discuss only a few of the built-in datatypes to get you started with SQL. I discuss all the datatypes and their usage in detail in Chapter 6.

CHAR(<size>)

The *CHAR* datatype is a fixed-length alphanumeric string, which has a maximum length in bytes (to specify length in characters, use the *CHAR* keyword inside parentheses along with a size; see Chapter 6). Data stored in *CHAR* columns is space-padded to fill the maximum length. Its size can range from a minimum of 1 byte to a maximum of 2,000 bytes. The default size is 1.

When you create a column using the *CHAR* datatype, the database will ensure that all data placed in this column has the defined length. If the data is shorter than the defined length, it is space-padded on the right to the specified length. If the data is longer, an error is raised.

VARCHAR2(<size>)

The *VARCHAR2* datatype is a variable-length alphanumeric string, which has a maximum length in bytes (to specify the length in characters, use the *CHAR* keyword inside parentheses along with a size; see Chapter 6). *VARCHAR2* columns require only the amount of space needed to store the data and can store up to 4,000 bytes. There is no default size for the *VARCHAR2* datatype. An empty *VARCHAR2*(2000) column takes up as much room in the database as an empty *VARCHAR2*(1) column.



The default size of a *CHAR* datatype is 1. For a *VARCHAR2* datatype, you must always specify the size.

The *VARCHAR2* and *CHAR* datatypes have different comparison rules for trailing spaces. With the *CHAR* datatype, trailing spaces are ignored. With the *VARCHAR2* datatype, trailing spaces are not ignored, and they sort higher than no trailing spaces. Here's an example:

CHAR datatype: 'Yo' = 'Yo '

VARCHAR2 datatype: 'Yo' < 'Yo '

NUMBER (<p>, <s>)

The *NUMBER* datatype stores numbers with a precision of <p> digits and a scale of <s> digits. The precision and scale values are optional. Numeric datatypes are used to store negative and positive integers, fixed-point numbers, and floating-point numbers. The precision can be between 1 and 38, and the scale has a range between -84 and 127. If the precision and scale are omitted, Oracle assumes the maximum of the range for both values.

You can have precision and scale digits in the integer part. The scale rounds the value after the decimal point to <s> digits. For example, if you define a column as *NUMBER*(5,2), the range of values you can store in this column is from -999.99 to 999.99; that is, 5 - 2 = 3 for the integer part, and the decimal part is rounded to two digits. Even if you do not include the decimal part for the value inserted, the maximum number you can store in a *NUMBER*(5,2) definition is 999.

Oracle will round numbers inserted into numeric columns with a scale smaller than the inserted number. For example, if a column were defined as *NUMBER*(4,2) and you specified a value of 12.125 to go into that column, the resulting number would be rounded to 12.13 before it was inserted into the column. If the value exceeds the precision, however, an Oracle error is returned. You cannot insert 123.1 into a column defined as *NUMBER*(4,2). Specifying the scale and precision does not force all inserted values to be a fixed length.

If the scale is negative, the number is rounded to the left of the decimal. Basically, a negative scale forces <s> number of zeros just to the left of the decimal.

If you specify a scale that is greater than the precision value, the precision defines the maximum number of digits to the right of the decimal point after the zeros. For example, if a column is defined as *NUMBER*(3,5), the range of values you can store is from -0.00999 to 0.00999; that is, it requires two zeros (<s>-<p>) after the decimal point and rounds the decimal part to three digits (<p>) after zeros. Table 1.3 shows several examples of how numeric data is stored with various definitions.

TABLE 1.3 Precision and Scale Examples

Value	Datatype	Stored Value	Explanation
123.2564	NUMBER	123.2564	The range and precision are set to the maximum, so the datatype can store any value.
1234.9876	NUMBER(6,2)	1234.99	Since the scale is only 2, the decimal part of the value is rounded to two digits.
12345.12345	NUMBER(6,2)	Error	The range of the integer part is only from -9999 to 9999.
123456	NUMBER(6,2)	Error	The precision is larger than specified; the range is only from -9999 to 9999.
1234.9876	NUMBER(6)	1235	The decimal part is rounded to the next integer.
123456.1	NUMBER(6)	123456	The decimal part is rounded.
12345.345	NUMBER(5,-2)	12300	The negative scale rounds the number <s> digits left to the decimal point. -2 rounds to hundreds.
1234567	NUMBER(5,-2)	1234600	Rounded to the nearest hundred.
12345678	NUMBER(5,-2)	Error	Outside the range; can have only five digits, excluding the two zeros representing hundreds, for a total of seven digits: $(s - (-p) = s + p = 5 + 2 = 7)$.
123456789	NUMBER(5,-4)	123460000	Rounded to the nearest 10,000.
1234567890	NUMBER(5,-4)	Error	Outside the range; can have only five digits, excluding the four trailing zeros.
12345.58	NUMBER(*, 1)	12345.6	The use of * in the precision specifies the default limit (38).
0.1	NUMBER(4,5)	Error	Requires a zero after the decimal point $(5 - 4 = 1)$.
0.01234567	NUMBER(4,5)	0.01235	Rounded to four digits after the decimal point and zero.

TABLE 1.3 Precision and Scale Examples *(continued)*

Value	Datatype	Stored Value	Explanation
0.09999	NUMBER(4,5)	0.09999	Stored as it is; only four digits after the decimal point and zero.
0.099996	NUMBER(4,5)	Error	Rounding this value to four digits after the decimal and zero results in 0.1, which is outside the range.

DATE

The *DATE* datatype is used to store date and time information. This datatype can be converted to other forms for viewing, but it has a number of special functions and properties that make date manipulation and calculations simple. The time component of the *DATE* datatype has a resolution of one second—no less. The *DATE* datatype occupies a storage space of 7 bytes. The following information is contained within each *DATE* datatype:

- Century
- Year
- Month
- Day
- Hour
- Minute
- Second

Date values are inserted or updated in the database by converting either a numeric value or a character value into a *DATE* datatype using the function *TO_DATE*. Oracle defaults the format to display the date as *DD-MON-YY*. This format shows that the default date must begin with a two-digit day, followed by a three-character abbreviation for the month, followed by a two-digit year. If you specify the date without including a time component, the time is defaulted to midnight, or 00:00:00 in military time. The *SYSDATE* function returns the current system date and time from the database server to which you’re currently connected.

TIMESTAMP [<precision>]

The *TIMESTAMP* datatype stores date and time information with fractional precision for seconds. The only difference between the *DATE* and *TIMESTAMP* datatypes is the ability to store fractional seconds up to a precision of nine digits. The default precision is 6 and can range from 0 to 9. Similar to the *SYSDATE* function, the *SYSTIMESTAMP* function returns the current system date and time, with fractional precision for seconds.

Operators and Literals

An *operator* is a manipulator that is applied to a data item in order to return a result. Special characters represent different operations in Oracle (+ represents addition, for example). Operators are commonly used in all programming environments, and you should already be familiar with the following operators, which may be classified into two types:

Unary operator A unary operator has only one operand. Examples are +2 and -5. They have the format <operator><operand>.

Binary operator A binary operator has two operands. Examples are 5+4 and 7*5. They have the format <operand1><operator><operand2>. You can insert spaces between the operand and operator to improve readability.

I'll now discuss the various types of operators available in Oracle.

Arithmetic Operators

Arithmetic operators operate on numeric values. Table 1.4 shows the various arithmetic operators in Oracle and how to use them.

TABLE 1.4 Arithmetic Operators

Operator	Purpose	Example
+ -	Unary operators: Use to represent positive or negative data item. For positive items, the + is optional.	-234.44
+	Addition: Use to add two data items or expressions.	2+4
-	Subtraction: Use to find the difference between two data items or expressions.	20.4-2
*	Multiplication: Use to multiply two data items or expressions.	5*10
/	Division: Use to divide a data item or expression with another.	8.4/2



Do not use two hyphens (--) to represent double negation; use a space or parentheses in between, as in -(-20). Two hyphens represent the beginning of a comment in SQL.

Concatenation Operator

The *concatenation operator* is used to concatenate or join two character (text) strings. The result of concatenation is another character string. Concatenating a zero-length string ('')

or a NULL with another string results in a string, not a NULL (NULL in Oracle 11g represents unknown or missing data). Two vertical bars (||) are used as the concatenation operator.

Here are two examples:

'Oracle11g' || 'Database' results in 'Oracle11gDatabase'.

'Oracle11g ' || 'Database' results in 'Oracle11g Database'.

Operator Precedence

If multiple operators are used in the same expression, Oracle evaluates them in the *order of precedence* set in the database engine. Operators with higher precedence are evaluated before operators with lower precedence. Operators with the same precedence are evaluated from left to right. Table 1.5 lists the precedence.

TABLE 1.5 SQL Operator Precedence

Precedence	Operator	Purpose
1	- +	Unary operators, negation
2	* /	Multiplication, division
3	+ -	Addition, subtraction, concatenation

Using parentheses changes the order of precedence. The innermost parenthesis is evaluated first. In the expression $1+2*3$, the result is 7, because $2*3$ is evaluated first and the result is added to 1. In the expression $(1+2)*3$, $1+2$ is evaluated first, and the result is multiplied by 3, giving 9.

Literals

Literals are values that represent a fixed value (constant). There are four types of literals:

- Text (or character)
- Numeric (integer and number)
- Datetime
- Interval

You can use literals within many of the SQL functions, expressions, and conditions.

Text Literals

A *text literal* must be enclosed in single quotation marks. Any character between the quotation marks is considered part of the text value. Oracle treats all text literals as though they were CHAR datatypes for comparison (blank padded). The maximum length of a text

literal is 4,000 bytes. Single quotation marks can be included in the literal text value by preceding it with another single quotation mark. Here are some examples of text literals:

```
'The Quick Brown Fox'
'That man''s suit is black'
'And I quote: "This will never do." '
'12-SEP-2001'
```

Alternatively, you can use Q or q quoting, which provides a range of delimiters. The syntax for using the Q/q quoting with a quote-delimiter text literal is as follows:

```
[Q|q]' <quote_delimiter> <text literal> <quote_delimiter>'
```

<quote_delimiter> is any character except a space, tab, or carriage return. The quote delimiter can be a single quotation mark, but make sure inside the text literal a single quotation mark is not immediately followed by another single quotation mark. If the opening quote delimiter is [or { or < or (, then the closing quote must be the corresponding] or } or > or). For all other quote delimiters, the opening quote delimiter must be the same as the closing quote delimiter. Here are some examples of text literals using the alternative quoting mechanism:

```
q'<The Quick Brown Fox>'
Q'#The Quick Brown Fox#'
q'{That man's suit is black}'
Q'(And I quote: "This will never do." )'
Q'"And I quote: "This will never do." "'
q'[12-SEP-2001]'
```

Numeric Literals

Integer literals can be any number of numerals, excluding a decimal separator and up to 38 digits long. Here are two examples:

- 24
- -456

Number and *floating-point literals* can include scientific notation, as well as digits and the decimal separator. E or e represents a number in scientific notation; the exponent can be in the range of -130 to 125. If the literal is followed by an f or F, it is treated as a BINARY_FLOAT datatype. If the literal is followed by a d or D, it is treated as a BINARY_DOUBLE datatype. Here are some examples:

- 24.0
- -345.65
- 23E-10

- 1.5f
- -34.567D
- -4d
- -4.0E+0

Datetime Literals

You can specify a date value as a string literal using the *datetime literals*. The most common methods to represent the datetime values are to use the conversion function `TO_DATE` or `TO_TIMESTAMP` with the appropriate format mask. For completeness of literals, I will discuss the datetime literals briefly.

The `DATE` literal uses the keyword `DATE` followed by the date value in single quotes, and the value must be specified in `YYYY-MM-DD` format with no time component. The time component will be defaulted to midnight (00:00:00). The following are examples of the `DATE` literal:

```
DATE '2008-03-24'
```

```
DATE '1999-12-31'
```

Similar to the `TIMESTAMP` datatype, the `TIMESTAMP` literal can be used to specify the year, month, date, hour, minute, second, and fractional second. You can also include time-zone data along with the `TIMESTAMP` literal. The time zone information can be specified using the UTC offset or using the time zone region name. The literal must be in the format `YYYY-MM-DD HH24:MI:SS TZ`. Here are some examples of the `TIMESTAMP` literal:

```
TIMESTAMP '2008-03-24 03:25:34.123'
```

```
TIMESTAMP '2008-03-24 03:25:34.123 -7:00'
```

```
TIMESTAMP '2008-03-24 03:25:34.123 US/Central'
```

```
TIMESTAMP '2008-03-24 03:25:34.123 US/Central CDT'
```

Interval Literals

Interval literals specify a period of time in terms of years and months or in terms of days and seconds. These literals correspond to the Oracle datatypes `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`. I'll discuss these datatypes in more detail in Chapter 6.

Writing Simple Queries

A *query* is a request for information from the database tables. Queries do not modify data; they read data from database tables and views. Simple queries are those that retrieve data from a single table or view. A table is used to store data and is stored in rows and columns. The basis of a query is the `SELECT` statement. The `SELECT` statement can be used to get data

from a single table or from multiple tables. Queries using multiple tables are discussed in later chapters.

Using the SELECT Statement

The SELECT statement is the most commonly used statement in SQL. It allows you to retrieve information already stored in the database. The statement begins with the keyword SELECT, followed by the column names whose data you want to query. You can select information either from all the columns (denoted by *) or from name-specific columns in the SELECT clause to retrieve data. The FROM clause provides the name of the table, view, or materialized view to use in the query. These objects are discussed in detail in later chapters. For simplicity, I will use tables for the rest of this chapter.

Let's use the JOBS table defined in the HR schema of the Oracle 11g sample database. You can use SQL*Plus tool to connect to the database as discussed earlier in the chapter. The JOBS table definition is provided in Table 1.6.

TABLE 1.6 JOBS Table Definition

Column Name	Datatype	Length
JOB_ID	VARCHAR2	10
JOB_TITLE	VARCHAR2	35
MIN_SALARY	NUMBER	6,0
MAX_SALARY	NUMBER	6,0

The simple form of a SELECT statement to retrieve all the columns and rows from the JOBS table is as follows (only part of output result set is shown here):

```
SQL> SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
...
IT_PROG	Programmer	4000	10000

MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

19 rows selected.



The keywords, column names, and table names are case insensitive. Only literals enclosed in single quotation marks are case sensitive in Oracle.

How do you list only the job title and minimum salary from this table? If you know the column names and the table name, writing the query is simple. Here, the column names are `JOB_TITLE` and `MIN_SALARY`, and the table name is `JOBS`. Execute the query by ending the query with a semicolon. In `SQL*Plus`, you can execute the query by entering a slash on a line by itself or by using the `RUN` command.

```
SQL> SELECT job_title, min_salary FROM jobs;
```

JOB_TITLE	MIN_SALARY
-----	-----
President	20000
Administration Vice President	15000
Administration Assistant	3000
Finance Manager	8200
Accountant	4200
Accounting Manager	8200
Public Accountant	4200
...	
Programmer	4000
Marketing Manager	9000
Marketing Representative	4000
Human Resources Representative	4000
Public Relations Representative	4500

19 rows selected.

Notice that the numeric column (`MIN_SALARY`) is aligned to the right and the character column (`JOB_TITLE`) is aligned to the left. Does it seem that the column heading `MIN_SALARY` should be more meaningful? Well, you can provide a *column alias* to appear in the query results.

Column Alias Names

The column alias name is defined next to the column name with a space or by using the keyword `AS`. If you want a space in the column alias name, you must enclose it in double quotation marks. The case is preserved only when the alias name is enclosed in double quotation marks; otherwise, the display will be uppercase. The following example demonstrates using an alias name for the column heading in the previous query:

```
SELECT job_title AS Title, min_salary AS "Minimum Salary"
FROM jobs;
```

TITLE	Minimum Salary
-----	-----
President	20000
Administration Vice President	15000
Administration Assistant	3000
Finance Manager	8200
Accountant	4200
Accounting Manager	8200
...	
Programmer	4000
Marketing Manager	9000
Marketing Representative	4000
Human Resources Representative	4000
Public Relations Representative	4500

19 rows selected.

In this listing, the column alias name `Title` appears in all capital letters because I did not enclose it in double quotation marks.



The asterisk (*) is used to select all columns in the table. This is useful when you do not know the column names or when you are too lazy to type all the column names.

Ensuring Uniqueness

The `DISTINCT` keyword (or `UNIQUE` keyword) following `SELECT` ensures that the resulting rows are unique. Uniqueness is verified against the complete row, not the first column. If you need to find the unique departments in the `EMPLOYEES` table, issue this query:

```
SELECT DISTINCT department_id
FROM employees;
```

DEPARTMENT_ID

```
-----
      100
      30

      20
      70
      90
     110
      50
      40
      80
      10
      60
```

12 rows selected.

To demonstrate that uniqueness is enforced across the row, let's do one more query using the `SELECT DISTINCT` clause. Notice `DEPARTMENT_ID` repeating for each `JOB_ID` value in the following example:

```
SELECT DISTINCT department_id, job_id
FROM employees;
```

```
DEPARTMENT_ID JOB_ID
-----
      110 AC_ACCOUNT
      90 AD_VP
      50 ST_CLERK
      80 SA_REP
      110 AC_MGR
... ..
      10 AD_ASST
      20 MK_REP
      40 HR_REP
      30 PU_MAN
```

20 rows selected.



`SELECT * FROM TAB;` shows all the tables and views in your schema. Don't be alarmed if you see a table name similar to `BIN$PJV23QpwQfu0zPN9uaXw+w==$0`. These are tables that belong to the Recycle Bin (or dropped tables). The tasks of creating tables and managing tables are discussed in Chapter 6.

The DUAL Table

The DUAL table is a dummy table available to all users in the database. It has one column and one row. The DUAL table is used to select system variables or to evaluate an expression. Here are few examples. The first query is to show the contents of the DUAL table.

```
SQL> SELECT * FROM dual;
```

```
DUMMY
```

```
-----
```

```
X
```

```
SQL> SELECT SYSDATE, USER FROM dual;
```

```
SYSDATE    USER
```

```
-----
```

```
18-SEP-07 HR
```

```
SQL> SELECT 'I'm ' || user || ' Today is ' || SYSDATE
2 FROM dual;
```

```
'I'M' || USER || 'TODAY IS' || SYSDATE
```

```
-----
```

```
I'm HR Today is 18-SEP-07
```



`SYSDATE` and `USER` are built-in functions that provide information about the environment. These functions are discussed in Chapter 2, "Using Single-Row Functions."

Limiting Rows

You can use the `WHERE` clause in the `SELECT` statement to limit the number of rows processed. Any logical conditions of the `WHERE` clause use the comparison operators. Rows

are returned or operated upon where the data satisfies the logical condition(s) of the **WHERE** clause. You can use column names or expressions in the **WHERE** clause, but not column alias names. The **WHERE** clause follows the **FROM** clause in the **SELECT** statement.

How do you list the employees who work for department 90? The following example shows how to limit the query to only the records belonging to department 90 by using a **WHERE** clause:

```
SELECT first_name || ' ' || last_name "Name", department_id
FROM   employees
WHERE  department_id = 90;
```

Name	DEPARTMENT_ID
-----	-----
Steven King	90
Neena Kochhar	90
Lex De Haan	90



You need not include the column names in the **SELECT** clause to use them in the **WHERE** clause.

You can use various operators in Oracle 11g in the **WHERE** clause to limit the number of rows.

Comparison Operators

Comparison operators compare two values or expressions and give a Boolean result of **TRUE**, **FALSE**, or **NULL**. The comparison operators include those that test for equality, inequality, less than, greater than, and value comparisons.

= (Equality)

The **=** operator tests for equality. The test evaluates to **TRUE** if the values or results of an expression on both sides of the operator are equal.

```
SELECT first_name || ' ' || last_name "Name", department_id
FROM   employees
WHERE  department_id = 90;
```

Name	DEPARTMENT_ID
-----	-----
Steven King	90
Neena Kochhar	90
Lex De Haan	90

!=, <>, or ^= (Inequality)

You can use any one of these three operators to test for inequality. The test evaluates to TRUE if the values on both sides of the operator do not match.

```
SELECT first_name || ' ' || last_name "Name", commission_pct
FROM   employees
WHERE  commission_pct != .35;
```

Name	COMMISSION_PCT
John Russell	.4
Karen Partners	.3
Alberto Errazuriz	.3
Gerald Cambrault	.3
...	...
Jack Livingston	.2
Kimberely Grant	.15
Charles Johnson	.1

32 rows selected.

< (Less Than)

The < operator evaluates to TRUE if the left side (expression or value) of the operator is less than the right side of the operator.

```
SELECT first_name || ' ' || last_name "Name", commission_pct
FROM   employees
WHERE  commission_pct < .15;
```

Name	COMMISSION_PCT
Mattea Marvins	.1
David Lee	.1
Sundar Ande	.1
Amit Banda	.1
Sundita Kumar	.1
Charles Johnson	.1

6 rows selected.

> (Greater Than)

The > operator evaluates to TRUE if the left side (expression or value) of the operator is greater than the right side of the operator.

```
SELECT first_name || ' ' || last_name "Name", commission_pct
FROM   employees
WHERE  commission_pct > .35;
```

Name	COMMISSION_PCT
John Russell	.4

<= (Less Than or Equal to)

The <= operator evaluates to TRUE if the left side (expression or value) of the operator is less than or equal to the right side of the operator.

```
SELECT first_name || ' ' || last_name "Name", commission_pct
FROM   employees
WHERE  commission_pct <= .15;
```

Name	COMMISSION_PCT
Oliver Tuvault	.15
Danielle Greene	.15
Mattea Marvins	.1
David Lee	.1
Sundar Ande	.1
Amit Banda	.1
William Smith	.15
Elizabeth Bates	.15
Sundita Kumar	.1
Kimberely Grant	.15
Charles Johnson	.1

11 rows selected.

>= (Greater Than or Equal to)

The >= operator evaluates to TRUE if the left side (expression or value) of the operator is greater than or equal to the right side of the operator.

```
SELECT first_name || ' ' || last_name "Name", commission_pct
FROM   employees
WHERE  commission_pct >= .35;
```

Name	COMMISSION_PCT
John Russell	.4
Janette King	.35
Patrick Sully	.35
Allan McEwen	.35

ANY or SOME

You can use the ANY or SOME operator to compare a value to each value in a list or subquery. The ANY and SOME operators always must be preceded by one of the following comparison operators: =, !=, <, >, <=, or >=.

```
SELECT first_name || ' ' || last_name "Name", department_id
FROM   employees
WHERE  department_id <= ANY (10, 15, 20, 25);
```

Name	DEPARTMENT_ID
Jennifer Whalen	10
Michael Hartstein	20
Pat Fay	20

ALL

You can use the ALL operator to compare a value to every value in a list or subquery. The ALL operator must always be preceded by one of the following comparison operators: =, !=, <, >, <=, or >=.

```
SELECT first_name || ' ' || last_name "Name", department_id
FROM   employees
WHERE  department_id >= ALL (80, 90, 100);
```

Name	DEPARTMENT_ID
Nancy Greenberg	100
Daniel Faviat	100
John Chen	100
Ismael Sciarra	100
Jose Manuel Urman	100
Luis Popp	100
Shelley Higgins	110
William Gietz	110

8 rows selected.

For all the comparison operators discussed, if one side of the operator is NULL, the result is NULL.

Logical Operators

Logical operators are used to combine the results of two comparison conditions (compound conditions) to produce a single result or to reverse the result of a single comparison. NOT, AND, and OR are the logical operators. When a logical operator is applied to NULL, the result is UNKNOWN. UNKNOWN acts similarly to FALSE; the only difference is that NOT FALSE is TRUE, whereas NOT UNKNOWN is also UNKNOWN.

NOT

You can use the NOT operator to reverse the result. It evaluates to TRUE if the operand is FALSE, and it evaluates to FALSE if the operand is TRUE. NOT returns NULL if the operand is NULL.

```
WHERE !(department_id >= 30)
      *
```

ERROR at line 3:

```
SELECT first_name, department_id
FROM   employees
WHERE  not (department_id >= 30);
```

FIRST_NAME	DEPARTMENT_ID
Jennifer	10
Michael	20
Pat	20

AND

The AND operator evaluates to TRUE if both operands are TRUE. It evaluates to FALSE if either operand is FALSE. Otherwise, it returns NULL.

```
SELECT first_name, salary
FROM   employees
WHERE  last_name = 'Smith'
AND    salary    > 7500;
```

FIRST_NAME	SALARY
Lindsey	8000

OR

The OR operator evaluates to TRUE if either operand is TRUE. It evaluates to FALSE if both operands are FALSE. Otherwise, it returns NULL.

```
SELECT first_name, last_name
FROM   employees
WHERE  first_name = 'Kelly'
OR     last_name  = 'Smith';
```

FIRST_NAME	LAST_NAME

Lindsey	Smith
William	Smith
Kelly	Chung

Logical Operator Truth Tables

The following tables are the truth tables for the three logical operators.

Table 1.7 is a truth table for the AND operator.

TABLE 1.7 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Table 1.8 is the truth table for the OR operator.

TABLE 1.8 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Table 1.9 is the truth table for the NOT operator.

TABLE 1.9 NOT Truth Table

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Other Operators

In the following sections, I will discuss all the operators that can be used in the WHERE clause of the SQL statement that were not discussed earlier.

IN and NOT IN

You can use the IN and NOT IN operators to test a membership condition. IN is equivalent to the =ANY operator, which evaluates to TRUE if the value exists in the list or the result set from a subquery. The NOT IN operator is equivalent to the !=ALL operator, which evaluates to TRUE if the value does not exist in the list or the result set from a subquery. The following examples demonstrate how to use these two operators:

```
SELECT first_name, last_name, department_id
FROM   employees
WHERE  department_id IN (10, 20, 90);
```

FIRST_NAME	LAST_NAME	DEPARTMENT_ID
-----	-----	-----
Steven	King	90
Neena	Kochhar	90
Lex	De Haan	90
Jennifer	Whalen	10
Michael	Hartstein	20
Pat	Fay	20

6 rows selected.

```
SELECT first_name, last_name, department_id
FROM   employees
WHERE  department_id NOT IN
      (10, 30, 40, 50, 60, 80, 90, 110, 100);
```

FIRST_NAME	LAST_NAME	DEPARTMENT_ID
Michael	Hartstein	20
Pat	Fay	20
Hermann	Baer	70

SQL>



When using the `NOT IN` operator, if any value in the list or the result returned from the subquery is `NULL`, the `NOT IN` condition is evaluated to `FALSE`. For example, `last_name not in ('Smith', 'Thomas', NULL)` evaluates to `last_name != 'Smith' AND last_name != 'Thomas' AND last_name != NULL`. Any comparison on a `NULL` value results in `NULL`. So, the previous condition does not return any row even though there may be some rows with `LAST_NAME` as `Smith` or `Thomas`.

BETWEEN

You can use the `BETWEEN` operator to test a range. `BETWEEN A AND B` evaluates to `TRUE` if the value is greater than or equal to `A` and less than or equal to `B`. If `NOT` is used, the result is the reverse. The following example lists all the employees whose salary is between \$5,000 and \$6,000:

```
SELECT first_name, last_name, salary
FROM   employees
WHERE  salary BETWEEN 5000 AND 6000;
```

FIRST_NAME	LAST_NAME	SALARY
Bruce	Ernst	6000
Kevin	Mourgos	5800
Pat	Fay	6000

EXISTS

The `EXISTS` operator is always followed by a subquery in parentheses. `EXISTS` evaluates to `TRUE` if the subquery returns at least one row. The following example lists the employees who work for the administration department. Here is an example of using `EXISTS`. Don't worry if you do not understand the SQL for now; subqueries are discussed in detail in Chapter 4, "Using Joins and Subqueries."

```
SELECT last_name, first_name, department_id
FROM   employees e
WHERE  EXISTS (select 1 FROM departments d
```



```
WHERE d.department_id = e.department_id
AND   d.department_name = 'Administration');
```

LAST_NAME	FIRST_NAME	DEPARTMENT_ID
Whalen	Jennifer	10

SQL>

IS NULL and IS NOT NULL

To find the NULL values or NOT NULL values, you need to use the IS NULL operator. The = or != operator will not work with NULL values. IS NULL evaluates to TRUE if the value is NULL. IS NOT NULL evaluates to TRUE if the value is not NULL. To find the employees who do not have a department assigned, use this query:

```
SELECT last_name, department_id
FROM   employees
WHERE  department_id IS NULL;
```

LAST_NAME	DEPARTMENT_ID
Grant	

```
SQL>
SELECT last_name, department_id
FROM   employees
WHERE  department_id = NULL;
```

no rows selected

LIKE

Using the LIKE operator, you can perform pattern matching. The pattern-search character % is used to match any character and any number of characters. The pattern-search character _ is used to match any single character. If you are looking for the actual character % or _ in the pattern search, you can include an escape character in the search string and notify Oracle using the ESCAPE clause.

The following query searches for all employees whose first name begins with Su and last name does not begin with S:

```
SELECT first_name, last_name
FROM   employees
WHERE  first_name LIKE 'Su%'
AND    last_name NOT LIKE 'S%';
```

FIRST_NAME	LAST_NAME

Sundar	Ande
Sundita	Kumar
Susan	Mavris

The following example looks for all `JOB_ID` values that begin with `AC_`. Since `_` is a pattern-matching character, you must qualify it with an escape character. Oracle does not have a default escape character.

```
SELECT job_id, job_title
FROM   jobs
WHERE  job_id like 'AC\_%' ESCAPE '\';
```

JOB_ID	JOB_TITLE

AC_MGR	Accounting Manager
AC_ACCOUNT	Public Accountant

Table 1.10 shows more examples of pattern matching.

TABLE 1.10 Pattern-Matching Examples

Pattern	Matches	Does Not Match
%SONI_1	SONIC1, ULTRASONI21	SONICS1, SONI315
_IME	TIME, LIME	IME, CRIME
\%SONI_1 ESCAPE '\'	%SONIC1, %SONI91	SONIC1, ULTRASONIC1
%ME_ _ _LE ESCAPE '\'	CRIME_FILE, TIME_POLE	CRIMESPILE, CRIME_ALE

Sorting Rows

The `SELECT` statement may include the `ORDER BY` clause to sort the resulting rows in a specific order based on the data in the columns. Without the `ORDER BY` clause, there is no guarantee that the rows will be returned in any specific order. If an `ORDER BY` clause is specified, by default the rows are returned by ascending order of the columns specified. If you need to sort the rows in descending order, use the keyword `DESC` next to the column name. You can specify the keyword `ASC` to explicitly state to sort in ascending order, although it is the

default. The ORDER BY clause follows the FROM clause and the WHERE clause in the SELECT statement.

To retrieve all employee names of department 90 from the EMPLOYEES table ordered by last name, use this query:

```
SELECT first_name || ' ' || last_name "Employee Name"
FROM   employees
WHERE  department_id = 90
ORDER BY last_name;
```

Employee Name

```
-----
Lex De Haan
Steven King
Neena Kochhar
SQL>
```

You can specify more than one column in the ORDER BY clause. In this case, the result set will be ordered by the first column in the ORDER BY clause, then the second, and so on. Columns or expressions not used in the SELECT clause can also be used in the ORDER BY clause. The following example shows how to use DESC and multiple columns in the ORDER BY clause:

```
SELECT first_name, hire_date, salary, manager_id mid
FROM   employees
WHERE  department_id IN (110,100)
ORDER BY mid ASC, salary DESC, hire_date;
```

FIRST_NAME	HIRE_DATE	SALARY	MID
Shelley	07-JUN-94	12000	101
Nancy	17-AUG-94	12000	101
Daniel	16-AUG-94	9000	108
John	28-SEP-97	8200	108
Jose Manuel	07-MAR-98	7800	108
Ismael	30-SEP-97	7700	108
Luis	07-DEC-99	6900	108
William	07-JUN-94	8300	205

8 rows selected.

SQL>



You can use column alias names in the ORDER BY clause.

If the DISTINCT keyword is used in the SELECT clause, you can use only those columns listed in the SELECT clause in the ORDER BY clause. If you have used any operators on columns in the SELECT clause, the ORDER BY clause also should use them. Here is an example:

```
SELECT DISTINCT 'Region ' || region_id
FROM   countries
ORDER BY region_id;
```

```
ORDER BY region_id
        *
```

```
ERROR at line 3:
ORA-01791: not a SELECTed expression
```

```
SELECT DISTINCT 'Region ' || region_id
FROM   countries
ORDER BY 'Region ' || region_id;
```

```
'REGION' || REGION_ID
```

```
-----
Region 1
Region 2
Region 3
Region 4
```

Not only can you use the column name or column alias to sort the result set of a query, but you can also sort the results by specifying the position of the column in the SELECT clause. This is useful if you have a lengthy expression in the SELECT clause and you need the results sorted on this value. The following example sorts the result set using positional values:

```
SELECT first_name, hire_date, salary, manager_id mid
FROM   employees
WHERE  department_id IN (110,100)
ORDER BY 4, 2, 3;
```

FIRST_NAME	HIRE_DATE	SALARY	MID
Shelley	07-JUN-94	12000	101

Nancy	17-AUG-94	12000	101
Daniel	16-AUG-94	9000	108
John	28-SEP-97	8200	108
Ismael	30-SEP-97	7700	108
Jose Manuel	07-MAR-98	7800	108
Luis	07-DEC-99	6900	108
William	07-JUN-94	8300	205

8 rows selected.



The ORDER BY clause cannot have more than 255 columns or expressions.

Sorting NULLs

By default, in an ascending-order sort, the NULL values appear at the bottom of the result set; that is, NULLs are sorted higher. For descending-order sorts, NULL values appear at the top of the result set—again, NULL values are sorted higher. You can change the default behavior by using the NULLS FIRST or NULLS LAST keyword, along with the column names (or alias names or positions). The following examples demonstrate how to use NULLS FIRST in an ascending sort:

```
SELECT last_name, commission_pct
FROM   employees
WHERE  last_name LIKE 'R%'
ORDER BY commission_pct ASC, last_name DESC;
```

LAST_NAME	COMMISSION_PCT
-----	-----
Russell	.4
Rogers	
Raphaely	
Rajs	

```
SELECT last_name, commission_pct
FROM   employees
WHERE  last_name LIKE 'R%'
ORDER BY commission_pct ASC NULLS FIRST, last_name DESC;
```

```
LAST_NAME          COMMISSION_PCT
-----
Rogers
Raphaely
Rajs
Russell            .4
SQL>
```

Why Do You Limit and Sort Rows?

The power of an RDBMS and SQL lies in getting exactly what you want from the database. The sample tables you considered under the HR schema are small, so even if you get all the information from the table, you can still find the specific data you’re seeking. But what if you have a huge transaction table with millions of rows?

You know how easy it is to look through a catalog in the library to find a particular book or to search through an alphabetical listing to find your name. When querying a large table, make sure you know what you want.

The WHERE clause lets you query for exactly what you’re looking for. The ORDER BY clause lets you sort rows. The following steps can be used as an approach to query data from single table:

- 1. Know the columns of the table. You can issue the DESCRIBE command to get the column names and datatype. Understand which column has what information.
- 2. Pick the column names you are interested in including in the query. Use these columns in the SELECT clause.
- 3. Identify the column or columns where you can limit the rows, or the columns that can show you only the rows of interest. Use these columns in the WHERE clause of the query, and supply the values as well as the appropriate operator.
- 4. If the query returns more than a few rows, you may be interested in having them sorted in a particular order. Specify the column names and the sorting order in the ORDER BY clause of the query.

Let’s consider a table named PURCHASE_ORDERS. First, use the DESCRIBE command to list the columns:

```
SQL> DESCRIBE purchase_orders
```

Name	Null?	Type
ORDER#	NOT NULL	NUMBER (16)
ORDER_DT	NOT NULL	DATE

```

CUSTOMER#          NOT NULL VARCHAR2 (12)
BACK_ORDER          CHAR (1)
ORD_STATUS          CHAR (1)
TOTAL_AMT           NOT NULL NUMBER (18,4)
SALES_TAX           NUMBER (12,2)

```

The objective of the query is to find the completed orders that do not have any sales tax. You want to see the order number and total amount of the order. The corresponding columns that appear in the SELECT clause are ORDER# and TOTAL_AMT. Since you're interested in only the rows with no sales tax in the completed orders, the columns to appear in the WHERE clause are SALES_TAX (checking for zero sales tax) and ORD_STATUS (checking for the completeness of the order, which is status code C). Since the query returns multiple rows, you want to order them by the order number. Notice that the SALES_TAX column can be NULL, so you want to make sure you get all rows that have a sales tax amount of zero or NULL.

```

SELECT order#, total_amt
FROM   purchase_orders
WHERE  ord_status = 'C'
AND    (sales_tax IS NULL
OR      sales_tax = 0)
ORDER BY order#;

```

An alternative is to use the NVL function to deal with the NULL values. This function is discussed in Chapter 2.

Using Expressions

An *expression* is a combination of one or more values, operators, and SQL functions that result in a value. The result of an expression generally assumes the datatype of its components. The simple expression 5+6 evaluates to 11 and assumes a datatype of NUMBER. Expressions can appear in the following clauses:

- The SELECT clause of queries
- The WHERE clause, ORDER BY clause, and HAVING clause
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

I will review the syntax of using these statements in later chapters.

You can include parentheses to group and evaluate expressions and then apply the result to the rest of the expression. When parentheses are used, the expression in the innermost

parentheses is evaluated first. Here is an example of a compound expression: $((2*4)/(3+1))*10$. The result of $2*4$ is divided by the result of $3+1$. Then the result from the division operation is multiplied by 10.

The CASE Expression

You can use the CASE expression to derive the IF...THEN...ELSE logic in SQL. Here is the syntax of the simple CASE expression:

```
CASE <expression>
WHEN <compare value> THEN <return value> ... ..
[ELSE <return value>]
END
```

The CASE expression begins with the keyword CASE and ends with the keyword END. The ELSE clause is optional. The maximum number of arguments in a CASE expression is 255. The following query displays a description for the REGION_ID column based on the value:

```
SELECT country_name, region_id,
       CASE region_id WHEN 1 THEN 'Europe'
                      WHEN 2 THEN 'America'
                      WHEN 3 THEN 'Asia'
                      ELSE 'Other' END Continent
FROM   countries
WHERE  country_name LIKE 'I%';
```

COUNTRY_NAME	REGION_ID	CONTINE
Israel	4	Other
India	3	Asia
Italy	1	Europe

SQL>

The other form of the CASE expression is the searched CASE, where the values are derived based on a condition. Oracle evaluates the conditions top to bottom; when a condition evaluates to true, the rest of the WHEN clauses are not evaluated. This version has the following syntax:

```
CASE
WHEN <condition> THEN <return value> ... ..
[ELSE <return value>]
END
```


The following example categorizes the salary as Low, Medium, and High using a searched CASE expression:

```
SELECT first_name, department_id, salary,
       CASE WHEN salary < 6000 THEN 'Low'
            WHEN salary < 10000 THEN 'Medium'
            WHEN salary >= 10000 THEN 'High' END Category
FROM   employees
WHERE  department_id <= 30
ORDER BY first_name;
```

FIRST_NAME	DEPARTMENT_ID	SALARY	CATEGORY
Alexander	30	3100	Low
Den	30	11000	High
Guy	30	2600	Low
Jennifer	10	4400	Low
Karen	30	2500	Low
Michael	20	13000	High
Pat	20	6000	Medium
Shelli	30	2900	Low
Sigal	30	2800	Low

9 rows selected.

Oracle uses the & (ampersand) character to substitute values at runtime. In the next section, I will discuss how to create SQL statements that can be used to get a different set of results based on values passed during execution time.

Finding the Current Sessions and Program Name

As a DBA you may have to query the V\$SESSION dictionary view to find the current sessions in the database. This view has several columns that show various information about the session; often the DBA is interested in finding out the username and which program is connecting to the database. If the DBA wants to find out what SQL is executed in the session, the SID and SERIAL# columns can be queried to enable tracing using the DBMS_TRACE package.

I'll review in this example how to query the V\$SESSION view using the simple SQL statements you learned in this chapter.

The following query may return several rows depending on the activity and number of users connected to the database:

```
SELECT username, sid, serial#, program
FROM v$session;
```

If you're using SQL*Plus, you may have to adjust the column width to fit the output in one line:

```
COLUMN program FORMAT a20
COLUMN username FORMAT a20
SELECT username, sid, serial#, program
FROM v$session;
```

USERNAME	SID	SERIAL#	PROGRAM
-----	-----	-----	-----
	118	6246	ORACLE.EXE (W000)
BTHOMAS	121	963	sqlplus.exe
DBSNMP	124	23310	emagent.exe
DBSNMP	148	608	emagent.exe
	150	1	ORACLE.EXE (FBDA)
	152	7	ORACLE.EXE (SMCO)
	155	1	ORACLE.EXE (MMNL)
	156	1	ORACLE.EXE (DIA0)
	158	1	ORACLE.EXE (MMON)
	159	1	ORACLE.EXE (RECO)
	164	1	ORACLE.EXE (MMAN)
... ..			(Output truncated)

As you can see, the background processes do not have usernames. To find out only the user sessions in the database, you can filter out the rows that do not have valid usernames:

```
SELECT username, sid, serial#, program
FROM v$session
WHERE username is NOT NULL;
```

If you're looking for specific information, you may want to add more filter conditions such as looking for a specific user or a specific program. The following SQL returns the rows in order of their session login time, with the most recent session on the top:

```
SELECT username, sid, serial#, program
FROM v$session
```

```
WHERE username is NOT NULL
ORDER BY logon_time;
```

USERNAME	SID	SERIAL#	PROGRAM
DBSNMP	148	608	emagent.exe
DBSNMP	124	23310	emagent.exe
BTHOMAS	121	963	sqlplus.exe
SCOTT	132	23	TOAD.EXE
SJACOB	231	32	discoverer.exe

Accepting Values at Runtime

To create an interactive SQL statement, you can define variables in the SQL statement. This allows the user to supply values at runtime, further enhancing the ability to reuse the SQL scripts. An ampersand (&) followed by a variable name prompts for and accepts values at runtime. For example, the following SELECT statement queries the DEPARTMENTS table based on the department number supplied at runtime.

```
SELECT department_name
FROM   departments
WHERE  department_id = &dept;
```

Enter value for dept: 10

old 3: WHERE DEPARTMENT_ID = &dept

new 3: WHERE DEPARTMENT_ID = 10

DEPARTMENT_NAME

Administration

1 row selected.

Using Substitution Variables

Suppose that you have defined DEPT as a variable in your script, but you want to avoid the prompt for the value at runtime. SQL*Plus prompts you for a value only when the variable is undefined. You can define a *substitution variable* in SQL*Plus using the DEFINE command

to provide a value. The variable will always have the CHAR datatype associated with it. Here is an example of defining a substitution variable:

```
SQL> DEFINE DEPT = 20
SQL> DEFINE DEPT
DEFINE DEPT                = "20" (CHAR)
SQL> LIST
   1  SELECT department_name
   2  FROM    departments
   3* WHERE   department_id = &DEPT
SQL> /
old   3: WHERE   DEPARTMENT_ID = &DEPT
new   3: WHERE   DEPARTMENT_ID = 20

DEPARTMENT_NAME
-----
Marketing

1 row selected.
SQL>
```



Using the DEFINE command without any arguments shows all the defined variables.

A . (dot) is used to append characters immediately after the substitution variable. The dot separates the variable name and the literal that follows immediately. If you need a dot to be part of the literal, provide two dots continuously. For example, the following query appends _REP to the user input when seeking a value from the JOBS table:

```
SQL> SELECT job_id, job_title FROM jobs
   2* WHERE   job_id = '&JOB._REP'
SQL> /
Enter value for job: MK
old   2: WHERE   JOB_ID = '&JOB._REP'
new   2: WHERE   JOB_ID = 'MK_REP'

JOB_ID      JOB_TITLE
-----
MK_REP      Marketing Representative

1 row selected.
SQL>
```

The old line with the variable and the new line with the substitution are displayed. You can turn off this display by using the command `SET VERIFY OFF`.

Saving a Variable for a Session

Consider the following SQL, saved to a file named `ex01.sql`. When you execute this script file, you will be prompted for the `COL1` and `COL2` values multiple times:

```
SQL> SELECT &COL1, &COL2
      2 FROM   &TABLE
      3 WHERE  &COL1 = '&VAL'
      4 ORDER BY &COL2
      5
SQL> SAVE ex01
Created file ex01.sql
SQL> @ex01
Enter value for col1: FIRST_NAME
Enter value for col2: LAST_NAME
old  1: SELECT &COL1, &COL2
new  1: SELECT FIRST_NAME, LAST_NAME
Enter value for table: EMPLOYEES
old  2: FROM   &TABLE
new  2: FROM   EMPLOYEES
Enter value for col1: FIRST_NAME
Enter value for val: John
old  3: WHERE  &COL1 = '&VAL'
new  3: WHERE  FIRST_NAME = 'John'
Enter value for col2: LAST_NAME
old  4: ORDER BY &COL2
new  4: ORDER BY LAST_NAME
```

FIRST_NAME	LAST_NAME
John	Chen
John	Russell
John	Seo

3 rows selected.

SQL>

The user can enter different or wrong values for each prompt. To avoid multiple prompts, use `&&` (double ampersand), where the variable is saved for the session.

To clear a defined variable, you can use the UNDEFINE command. Let's edit the `ex01.sql` file to make it look like this:

```
SELECT &&COL1, &&COL2
FROM   &TABLE
WHERE  &COL1 = '&VAL'
ORDER BY &COL2
/
Enter value for col1: first_name
Enter value for col2: last_name
old   1: SELECT &&COL1, &&COL2
new   1: SELECT first_name, last_name
Enter value for table: employees
old   2: FROM &TABLE
new   2: FROM employees
Enter value for val: John
old   3: WHERE &COL1 = '&VAL'
new   3: WHERE first_name = 'John'
old   4: ORDER BY &COL1
new   4: ORDER BY first_name
```

FIRST_NAME	LAST_NAME
John	Chen
John	Russell
John	Seo

```
UNDEFINE COL1 COL2
```

Using Positional Notation for Variables

Instead of variable names, you can use positional notation, where each variable is identified by `&1`, `&2`, and so on. The values are assigned to the variables by position. Do this by putting an ampersand (&), followed by a numeral, in place of a variable name. Consider the following query:

```
SQL> SELECT department_name, department_id
      2 FROM   departments
      3 WHERE  &1 = &2;
Enter value for 1: DEPARTMENT_ID
Enter value for 2: 10
old   3: WHERE  &1 = &2
new   3: WHERE  DEPARTMENT_ID = 10
```

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10

1 row selected.

SQL>

If you save the SQL as a script file, you can submit the substitution-variable values while invoking the script (as command-line arguments). Each time you run this command file, `START` replaces each `&1` in the file with the first value (called an *argument*) after `START filename`, then replaces each `&2` with the second value, and so forth. Here is an example of saving and running the previous query:

SQL> SAVE ex02

Created file ex02.sql

SQL> SET VERIFY OFF

SQL> @ex02 department_id 20

DEPARTMENT_NAME	DEPARTMENT_ID
Marketing	20

1 row selected.

SQL>

Although I did not specify two ampersands for positional substitution variables, SQL*Plus keeps the values of these variables for the session (since we passed the values as parameters to a script file). Next time you run any script with positional substitution variables, Oracle uses these values to execute the script.

Summary

This chapter started off with reviewing the fundamentals of SQL. You also saw an overview of SQL*Plus in this chapter. SQL*Plus is Oracle's native tool to interact with the database. You got a quick introduction to the Oracle datatypes, operators, and literals. You learned to write simple queries using the `SELECT` statement. You also learned to use the `WHERE` clause and the `ORDER BY` clause in this chapter.

The `CHAR` and `VARCHAR2` datatypes are used to store alphanumeric information. The `NUMBER` datatype is used to store any numeric value. Date values can be stored using the `DATE` or `TIMESTAMP` datatypes. Oracle has a wide range of operators: arithmetic, concatenation, comparison, membership, logical, pattern matching, range, existence, and `NULL` checking. The `CASE` expression is used to bring conditional logic to SQL.

SQL*Plus supports all SQL statements and has its own formatting and enhancement commands. Using this tool, you can produce interactive SQL statements and formatted reports. SQL*Plus is the command-line interface to the database widely used by DBAs. SQL*Plus has its own buffer where SQL statements are buffered. You can edit the buffer using SQL*Plus editing commands. The DESCRIBE command is used to get information on a table, view, function, or procedure. Multiple SQL and SQL*Plus commands can be stored in a file and can be executed as a unit. Such files are called script files.

Data in the Oracle database is managed and accessed using SQL. A SELECT statement is the basic form of querying or reading records from the database table. You can limit or filter the rows using the WHERE clause. You can use the AND and OR logical operators to join multiple filter conditions. The ORDER BY clause is used to sort the result set in a particular order. You can use an ampersand (&) character to substitute a value at runtime.

Exam Essentials

Understand the operators. Know the various operators that can be used in queries. The parentheses around an expression change the precedence of the operators.

Understand the WHERE clause. The WHERE clause specifies a condition to limit the number of rows returned. You cannot use column alias names in this clause.

Understand the ORDER BY clause. The ORDER BY clause is used to sort the result set from a query. You can specify ascending order or descending order for the sort. Ascending order is the default. Also know that column alias names can be used in the ORDER BY clause. You can also specify columns by their position.

Know how to specify string literals using the Q/q operator. You can use the Q or q operator to specify the quote delimiters in string literals. Understand the difference between using the C, <, {, and [characters and other delimiters.

Know the order of clauses in the SELECT statement. The SELECT statement must have a FROM clause. The WHERE clause, if it exists, should follow the FROM clause and precede the ORDER BY clause.

Know the use of the DUAL table. The DUAL table is a dummy table in Oracle with one column and one row. This table is commonly used to get the values of system variables such as SYSDATE or USER.

Know the characters used for pattern matching. The % character is used to match zero or more characters. The _ character is used to match one, and only one, character. The SQL operator used with a pattern-matching character is LIKE.

Know the sort order of NULL values in queries with ORDER BY clause. By default, in an ascending-order sort, the NULL values appear at the bottom of the result set; that is, NULLs are sorted higher. For descending-order sorts, NULL values appear at the top of the result set—again, NULL values are sorted higher.

Review Questions

1. You issue the following query:

```
SELECT salary "Employee Salary"  
FROM employees;
```

How will the column heading appear in the result?

- A. EMPLOYEE SALARY
- B. EMPLOYEE_SALARY
- C. Employee Salary
- D. employee_salary

2. The EMP table is defined as follows:

Column	Datatype	Length
EMPNO	NUMBER	4
ENAME	VARCHAR2	30
SALARY	NUMBER	14,2
COMM	NUMBER	10,2
DEPTNO	NUMBER	2

You perform the following two queries:

- 1.

```
SELECT empno enumber,  ename  
FROM emp ORDER BY 1;
```
- 2.

```
SELECT empno,  ename  
FROM emp ORDER BY  empno ASC;
```

Which of the following is true?

- A. Statements 1 and 2 will produce the same result in data.
- B. Statement 1 will execute; statement 2 will return an error.
- C. Statement 2 will execute; statement 1 will return an error.
- D. Statements 1 and 2 will execute but produce different results.

3. You issue the following SELECT statement on the EMP table shown in question 2.

```
SELECT (200+((salary*0.1)/2)) FROM emp;
```

What will happen to the result if all the parentheses are removed?

- A. No difference, because the answer will always be NULL.
 - B. No difference, because the result will be the same.
 - C. The result will be higher.
 - D. The result will be lower.
4. In the following SELECT statement, which component is a literal? (Choose all that apply.)

```
SELECT 'Employee Name: ' || ename  
FROM emp where deptno = 10;
```

- A. 10
 - B. ename
 - C. Employee Name:
 - D. ||
5. When you try to save 34567.2255 into a column defined as NUMBER(7,2), what value is actually saved?
- A. 34567.00
 - B. 34567.23
 - C. 34567.22
 - D. 3456.22
6. What is the default display length of the DATE datatype column?
- A. 18
 - B. 9
 - C. 19
 - D. 6
7. What will happen if you query the EMP table shown in question 2 with the following?
- ```
SELECT empno, DISTINCT ename, salary FROM emp;
```
- A. EMPNO, unique values of ENAME, and then SALARY are displayed.
  - B. EMPNO and unique values of the two columns, ENAME and SALARY, are displayed.
  - C. DISTINCT is not a valid keyword in SQL.
  - D. No values will be displayed because the statement will return an error.
8. Which clause in a query limits the rows selected?
- A. ORDER BY
  - B. WHERE
  - C. SELECT
  - D. FROM

9. The following listing shows the records of the EMP table:

| EMPNO | ENAME  | SALARY | COMM  | DEPTNO |
|-------|--------|--------|-------|--------|
| 7369  | SMITH  | 800    |       | 20     |
| 7499  | ALLEN  | 1600   | 300   | 30     |
| 7521  | WARD   | 1250   | 500   | 30     |
| 7566  | JONES  | 2975   |       | 20     |
| 7654  | MARTIN | 1250   | 1400  | 30     |
| 7698  | BLAKE  | 2850   |       | 30     |
| 7782  | CLARK  | 2450   | 24500 | 10     |
| 7788  | SCOTT  | 3000   |       | 20     |
| 7839  | KING   | 5000   | 50000 | 10     |
| 7844  | TURNER | 1500   | 0     | 30     |
| 7876  | ADAMS  | 1100   |       | 20     |
| 7900  | JAMES  | 950    |       | 30     |
| 7902  | FORD   | 3000   |       | 20     |
| 7934  | MILLER | 1300   | 13000 | 10     |

When you issue the following query, which value will be displayed in the first row?

```
SELECT empno
FROM emp
WHERE deptno = 10
ORDER BY ename DESC;
```

- A. MILLER
  - B. 7934
  - C. 7876
  - D. No rows will be returned because ename cannot be used in the ORDER BY clause.
10. Refer to the listing of records in the EMP table in question 9. How many rows will the following query return?

```
SELECT * FROM emp WHERE ename BETWEEN 'A' AND 'C'
```

- A. 4
- B. 2
- C. A character column cannot be used in the BETWEEN operator.
- D. 3

11. Refer to the EMP table in question 2. When you issue the following query, which line has an error?

```
1. SELECT empno "Enumber", ename "EmpName"
2. FROM emp
3. WHERE deptno = 10
4. AND "Enumber" = 7782
5. ORDER BY "Enumber";
```

- A. 1
- B. 5
- C. 4
- D. No error; the statement will finish successfully.

12. You issue the following query:

```
SELECT empno, ename
FROM emp
WHERE empno = 7782 OR empno = 7876;
```

Which other operator can replace the OR condition in the WHERE clause?

- A. IN
- B. BETWEEN .. AND ..
- C. LIKE
- D. <=
- E. >=

13. The following are clauses of the SELECT statement:

```
1. WHERE
2. FROM
3. ORDER BY
```

In which order should they appear in a query?

- A. 1, 3, 2
- B. 2, 1, 3
- C. 2, 3, 1
- D. The order of these clauses does not matter.

14. Which statement searches for PRODUCT\_ID values that begin with DI\_ from the ORDERS table?

```
A. SELECT * FROM ORDERS
 WHERE PRODUCT_ID = 'DI%';

B. SELECT * FROM ORDERS
 WHERE PRODUCT_ID LIKE 'DI_' ESCAPE '\';

C. SELECT * FROM ORDERS
 WHERE PRODUCT_ID LIKE 'DI_%' ESCAPE '\';
```

- D. `SELECT * FROM ORDERS  
WHERE PRODUCT_ID LIKE 'DI\_ ' ESCAPE '\';`
- E. `SELECT * FROM ORDERS  
WHERE PRODUCT_ID LIKE 'DI_%' ESCAPE '\';`
15. COUNTRY\_NAME and REGION\_ID are valid column names in the COUNTRIES table. Which one of the following statements will execute without an error?
- A. `SELECT country_name, region_id,  
CASE region_id = 1 THEN 'Europe',  
      region_id = 2 THEN 'America',  
      region_id = 3 THEN 'Asia',  
      ELSE 'Other' END Continent  
FROM countries;`
- B. `SELECT country_name, region_id,  
CASE (region_id WHEN 1 THEN 'Europe',  
      WHEN 2 THEN 'America',  
      WHEN 3 THEN 'Asia',  
      ELSE 'Other') Continent  
FROM countries;`
- C. `SELECT country_name, region_id,  
CASE region_id WHEN 1 THEN 'Europe'  
      WHEN 2 THEN 'America'  
      WHEN 3 THEN 'Asia'  
      ELSE 'Other' END Continent  
FROM countries;`
- D. `SELECT country_name, region_id,  
CASE region_id WHEN 1 THEN 'Europe'  
      WHEN 2 THEN 'America'  
      WHEN 3 THEN 'Asia'  
      ELSE 'Other' Continent  
FROM countries;`
16. Which special character is used to query all the columns from the table without listing each column by name?
- A. %
- B. &
- C. @
- D. \*

17. The EMPLOYEE table has the following data:

| EMP_NAME | HIRE_DATE | SALARY |
|----------|-----------|--------|
| SMITH    | 17-DEC-90 | 800    |
| ALLEN    | 20-FEB-91 | 1600   |
| WARD     | 22-FEB-91 | 1250   |
| JONES    | 02-APR-91 | 5975   |
| WARDEN   | 28-SEP-91 | 1250   |
| BLAKE    | 01-MAY-91 | 2850   |

What will be the value in the first row of the result set when the following query is executed?

```
SELECT hire_date FROM employee
ORDER BY salary, emp_name;
```

- A. 02-APR-91
  - B. 17-DEC-90
  - C. 28-SEP-91
  - D. The query is invalid, because you cannot have a column in the ORDER BY clause that is not part of the SELECT clause.
18. Which SQL statement will query the EMPLOYEES table for FIRST\_NAME, LAST\_NAME, and SALARY of all employees in DEPARTMENT\_ID 40 in the alphabetical order of last name?
- A. 

```
SELECT first_name last_name salary
FROM employees
ORDER BY last_name
WHERE department_id = 40;
```
  - B. 

```
SELECT first_name, last_name, salary
FROM employees
ORDER BY last_name ASC
WHERE department_id = 40;
```
  - C. 

```
SELECT first_name last_name salary
FROM employees
WHERE department_id = 40
ORDER BY last_name ASC;
```
  - D. 

```
SELECT first_name, last_name, salary
FROM employees
WHERE department_id = 40
ORDER BY last_name;
```
  - E. 

```
SELECT first_name, last_name, salary
FROM TABLE employees
WHERE department_id IS 40
ORDER BY last_name ASC;
```
19. When doing pattern matching using the LIKE operator, which character is used as the default escape character by Oracle?
- A. |
  - B. /
  - C. \
  - D. There is no default escape character in Oracle.

20. Column alias names cannot be used in which clause?

- A. SELECT clause
- B. WHERE clause
- C. ORDER BY clause
- D. None of the above

21. What is wrong with the following statements submitted in SQL\*Plus?

```
DEFINE V_DEPTNO = 20
SELECT LAST_NAME, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID = V_DeptNo;
```

- A. Nothing is wrong. The query lists the employee name and salary of the employees who belong to department 20.
- B. The DEFINE statement declaration is wrong.
- C. The substitution variable is not preceded with the & character.
- D. The substitution variable in the WHERE clause should be V\_DEPTNO instead of V\_DeptNo.

22. Which two statements regarding substitution variables are true?

- A. *&variable* is defined by SQL\*Plus, and its value will be available for the duration of the session.
- B. *&&variable* is defined by SQL\*Plus, and its value will be available for the duration of the session.
- C. *&n* (where *n* is a any integer) variables are defined by SQL\*Plus when values are passed in as arguments to the script, and their values will be available for the duration of the session.
- D. *&&variable* is defined by SQL\*Plus, and its value will be available only for every reference to that variable in the current SQL.

23. Look at the data in table PRODUCTS. Which SQL will list the items on the BL shelves? (Show the result with the most available quantity at the top row.)

| PRODUCT_ID | PRODUCT_NAME | SHELF | AVAILABLE_QTY |
|------------|--------------|-------|---------------|
| 1001       | CREST        | BL36  | 354           |
| 1002       | COLGATE      | BL36  | 54            |
| 1003       | AQUAFRESH    | BL37  | 43            |
| 2002       | SUNNY-D      | LA21  | 53            |
| 2003       | CAPRISUN     | LA22  | 45            |

- A. `SELECT * FROM products  
WHERE shelf like '%BL'  
ORDER BY available_qty SORT DESC;`
- B. `SELECT * FROM products  
WHERE shelf like 'BL%';`
- C. `SELECT * FROM products  
WHERE shelf = 'BL%'  
ORDER BY available_qty DESC;`
- D. `SELECT * FROM products  
WHERE shelf like 'BL%'  
ORDER BY available_qty DESC;`
- E. `SELECT * FROM products  
WHERE shelf like 'BL%'  
ORDER BY available_qty SORT;`

24. The EMP table has the following data:

| EMPNO | ENAME  | SAL  | COMM |
|-------|--------|------|------|
| 7369  | SMITH  | 800  |      |
| 7499  | ALLEN  | 1600 | 300  |
| 7521  | WARD   | 1250 | 500  |
| 7566  | JONES  | 2975 |      |
| 7654  | MARTIN | 1250 | 1400 |
| 7698  | BLAKE  | 2850 |      |
| 7782  | CLARK  | 2450 |      |
| 7788  | SCOTT  | 3000 |      |
| 7839  | KING   | 5000 |      |
| 7844  | TURNER | 1500 | 0    |
| 7876  | ADAMS  | 1100 |      |
| 7900  | JAMES  | 950  |      |
| 7902  | FORD   | 3000 |      |
| 7934  | MILLER | 1300 |      |

Consider the following two SQL statements:

1. `SELECT empno, ename, sal, comm  
FROM emp WHERE comm IN (0, NULL);`
  2. `SELECT empno, ename, sal, comm  
FROM emp WHERE comm = 0 OR comm IS NULL;`
- A. 1 and 2 will produce the same result.
  - B. 1 will error; 2 will work fine.
  - C. 1 and 2 will produce different results.
  - D. 1 and 2 will work but will not return any rows.



# Answers to Review Questions

1. C. Column alias names enclosed in quotation marks will appear as typed. Spaces and mixed case appear in the column alias name only when the alias is enclosed in double quotation marks.
2. A. Statements 1 and 2 will produce the same result. You can use the column name, column alias, or column position in the ORDER BY clause. The default sort order is ascending. For a descending sort, you must explicitly specify that order with the DESC keyword.
3. B. In the arithmetic evaluation, multiplication and division have precedence over addition and subtraction. Even if you do not include the parentheses, `salary*0.1` will be evaluated first. The result is then divided by 2, and its result is added to 200.
4. A, C. Character literals in the SQL statement are enclosed in single quotation marks. Literals are concatenated using `||`. `Employee Name:` is a character literal, and 10 is a numeric literal.
5. B. Since the numeric column is defined with precision 7 and scale 2, you can have five digits in the integer part and two digits after the decimal point. The digits after the decimal are rounded.
6. B. The default display format of DATE column is DD-MON-YY, whose length is 9.
7. D. DISTINCT is used to display a unique result row, and it should follow immediately after the keyword SELECT. Uniqueness is identified across the row, not a single column.
8. B. The WHERE clause is used to limit the rows returned from a query. The WHERE clause condition is evaluated, and rows are returned only if the result is TRUE. The ORDER BY clause is used to display the result in certain order.
9. B. There are three records belonging to DEPTNO 10: EMPNO 7934 (MILLER), 7839 (KING), and 7782 (CLARK). When you sort their names by descending order, MILLER is the first row to display. You can use alias names and columns that are not in the SELECT clause in the ORDER BY clause.
10. D. Here, a character column is compared against a string using the BETWEEN operator, which is equivalent to `ename >= 'A' AND ename <= 'C'`. The name CLARK will not be included in this query, because 'CLARK' is > 'C'.
11. C. Column alias names cannot be used in the WHERE clause. They can be used in the ORDER BY clause.
12. A. The IN operator can be used. You can write the WHERE clause as `WHERE empno IN (7782, 7876);`.
13. B. The FROM clause appears after the SELECT statement, followed by WHERE and ORDER BY clauses. The FROM clause specifies the table names, the WHERE clause limits the result set, and the ORDER BY clause sorts the result.

14. C. Since `_` is a special pattern-matching character, you need to include the `ESCAPE` clause in `LIKE`. The `%` character matches any number of characters including 0, and `_` matches a single character.
15. C. A `CASE` expression begins with the keyword `CASE` and ends with the keyword `END`.
16. D. An asterisk (`*`) is used to denote all columns in a table.
17. B. The default sorting order for a numeric column is ascending. The columns are sorted first by salary and then by name, so the row with the lowest salary is displayed first. It is perfectly valid to use a column in the `ORDER BY` clause that is not part of the `SELECT` clause.
18. D. In the `SELECT` clause, the column names should be separated by commas. An alias name may be provided for each column with a space or using the keyword `AS`. The `FROM` clause should appear after the `SELECT` clause. The `WHERE` clause appears after the `FROM` clause. The `ORDER BY` clause comes after the `WHERE` clause.
19. D. There is no default escape character in Oracle for pattern matching. If your search includes pattern-matching characters such as `_` or `%`, define an escape character using the `ESCAPE` keyword in the `LIKE` operator.
20. B. Column alias names cannot be used in the `WHERE` clause of the SQL statement. In the `ORDER BY` clause, you can use the column name or alias name, or you can indicate the column by its position in the `SELECT` clause.
21. C. The query will return an error, because the substitution variable is used without an ampersand (`&`) character. In this query, Oracle treats `V_DEPTNO` as another column name from the table and returns an error. Substitution variables are not case sensitive.
22. B, C. When a variable is preceded by double ampersands, `SQL*Plus` defines that variable. Similarly, when you pass values to a script using `START script_name arguments`, `SQL*Plus` defines those variables. Once a variable is defined, its value will be available for the duration of the session or until you use `UNDEFINE variable`.
23. D. `%` is the wild character to pattern-match for any number of characters. Option A is almost correct, except for the `SORT` keyword in the `ORDER BY` clause, which will produce an error since it is not a valid syntax. Option B will produce results but will sort them in the order you want. Option C will not return any rows because `LIKE` is the operator for pattern matching, not `=`. Option E has an error similar to Option A.
24. C. In the first SQL, the `comm IN (0, NULL)` will be treated as `comm = 0 OR comm = NULL`. For all `NULL` comparisons, you should use `IS NULL` instead of `= NULL`. The first SQL will return only one row where `comm = 0`, whereas the second SQL will return all the rows that have `comm = NULL` as well as `comm = 0`.