CHAPTER 1

Core SQL

Karen Morton

Whether you're relatively new to writing SQL or you've been writing it for years, learning to write "good" SQL is a process that requires a strong foundation knowledge of core syntax and concepts. This chapter provides a review of the core concepts of the SQL language and its capabilities along with descriptions of the common SQL commands with which you should already be familiar. For those of you who have worked with SQL previously and have a good grasp on the basics, this will be a brief refresher, and it will prepare you for the more detailed treatment of SQL we'll cover in the chapters ahead. If you're a new SQL user, you may want to read *Beginning Oracle SQL* first to make sure you're comfortable with the basics. Either way, Chapter 1 is intended to "level set" you with a whirlwind tour of the five core SQL statements and provide a quick review of the tool we'll be using to execute SQL, SQL*Plus.

The SQL Language

The SQL language was originally developed in the 1970s by IBM and called Structured English QUEry Language, or SEQUEL. The language was based on the model for relational database management systems (RDBMS) developed by E.F. Codd in 1969. The acronym was later shortened to SQL due to a trademark dispute. In 1986, ANSI adopted SQL as a standard, and in 1987, ISO did so as well. A piece of not-so-common knowledge is that the official pronunciation of the language was declared to be "ess queue ell" by ANSI. Most people, including me, still use the *sequel* pronunciation just because it flows a bit easier linguistically.

The purpose of SQL is to simply provide an interface to the database, in our case, Oracle. Every SQL statement is a command, or instruction, to the database. It differs from other programming languages like C and Java in that it is intended to process data in sets, not individual rows. The language also doesn't require that you provide instructions on how to navigate to the data—that happens transparently under the covers. But, as you'll see in the chapters ahead, knowing about your data and how and where it is stored can be very important if you want to write efficient SQL in Oracle.

While there are minor differences in how vendors (like Oracle, IBM and Microsoft) implement the core functionality of SQL, the skills you learn in one database will transfer to another. You will be able to use basically the same SQL statements to query, insert, update, and delete data and create, alter, and drop objects regardless of the database vendor.

Although SQL is the standard for use with various RDBMS, it is not particularly relational in practice. I'll expand on this a bit later in the book; I would also recommend that you read C.J. Date's book entitled *SQL and Relational Theory* for a more detailed review. Keep in mind that the SQL language doesn't always follow the relational model precisely—it doesn't implement some elements of the relational model at all while implementing other elements improperly. The fact remains that

K. Morton et al., *Pro Oracle SQL*

since SQL is based on this model you must not only understand SQL but you must understand the relational model as well in order to write SQL as correctly and efficiently as possible.

Interfacing to the Database

Numerous ways have been developed over the years for transmitting SQL to a database and getting results back. The native interface to the Oracle database is the Oracle Call Interface (OCI). The OCI powers the queries that are sent by the Oracle kernel internally to the database. You use the OCI anytime you use one of Oracle's tools like SQL*Plus or SQL Developer. Various other Oracle tools like SQL*Loader, Data Pump, and Real Application Testing (RAT) use OCI as well as language specific interfaces such as Oracle JDBC-OCI, ODP.Net, Oracle Precompilers, Oracle ODBC, and the Oracle C++ Call Interface (OCCI) drivers.

When you use programming languages like COBOL or C, the statements you write are known as Embedded SQL statements and are preprocessed by a SQL preprocessor before the application program is compiled. Listing 1-1 shows an example of a SQL statement that could be used within a C/C++ block.

Listing 1-1. *Embedded SQL Statement Used Within C/C++ Block*

```
{
        int a;
        /* ... */
        EXEC SQL SELECT salary INTO :a
                    FROM hr.employees
                   WHERE employee_id = 108;
        /* ... */
        printf("The salary is %d\n", a);
        /* ... */
}
```

Other tools, like SQL*Plus and SQL Developer, are interactive tools. You enter and execute commands, and the output is displayed back to you. Interactive tools don't require you to explicitly compile your code before running it. You simply enter the command you wish to execute. Listing 1-2 shows an example of using SQL*Plus to execute a statement.

Listing 1-2. *Using SQL*Plus to Execute a SQL Statement*

```
SQL> select  salary
  2  from    hr.employees
  3  where   employee_id = 108;

         SALARY
---------------
          12000
```

In this book, we'll use SQL*Plus for our example listings just for consistency, but keep in mind that whatever method or tool you use to enter and execute SQL statements, everything ultimately goes through the OCI. The bottom line is that the tool you use doesn't matter, the native interface is the same for all.

Review of SQL*Plus

SQL*Plus is a command line tool provided with every Oracle installation regardless of platform (Windows, Unix). It is used to enter and execute SQL commands and to display the resulting output in a text-only environment. The tool allows you to enter and edit commands, save and execute commands individually or via script files, and display the output in nicely formatted report form. To start SQL*Plus you simply start `sqlplus` from your host's command prompt.

Connect to a Database

There are multiple ways to connect to a database from SQL*Plus. Before you can connect however, you will likely need to have entries for the databases you will need to connect to entered in the `$ORACLE_HOME/network/admin/tnsnames.ora` file. Two common ways are to supply your connection information when you start SQL*Plus, as shown in Listing 1-3; the other is to use the SQL*Plus `connect` command after SQL*Plus starts, as shown in Listing 1-4.

Listing 1-3. *Connecting to SQL*Plus from the Windows Command Prompt*

```
E:\pro_oracle_sql>sqlplus hr@ora11r2

SQL*Plus: Release 11.2.0.1.0 - Production on Sun Jun 6 11:22:24 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.
Enter password:

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining and Real Appliation Testing
options

SQL>
```

To start SQL*Plus without being prompted to login to a database, you can start SQL*Plus with the `/nolog` option.

Listing 1-4. *Connecting to SQL*Plus and Logging into the Database from The SQL> Prompt*

```
E:\pro_oracle_sql>sqlplus /nolog

SQL*Plus: Release 11.2.0.1.0 - Production on Sun Jun 6 11:22:24 2010

Copyright (c) 1982, 2010, Oracle.  All rights reserved.

SQL> connect hr@ora11r2
Enter password:
Connected.
SQL>
```

Configuring the SQL*Plus environment

SQL*Plus has numerous commands that allow you to customize the working environment and display options. Listing 1-5 shows the SQL*Plus commands available after entering the SQL*Plus `help index` command at the `SQL>` prompt.

Listing 1-5. *SQL*Plus Command List*

```
SQL> help index

Enter Help [topic] for help.

@               COPY            PAUSE                        SHUTDOWN
@@              DEFINE          PRINT                        SPOOL
/               DEL             PROMPT                       SQLPLUS
ACCEPT          DESCRIBE        QUIT                         START
APPEND          DISCONNECT      RECOVER                      STARTUP
ARCHIVE LOG     EDIT            REMARK                       STORE
ATTRIBUTE       EXECUTE         REPFOOTER                    TIMING
BREAK           EXIT            REPHEADER                    TTITLE
BTITLE          GET             RESERVED WORDS (SQL)         UNDEFINE
CHANGE          HELP            RESERVED WORDS (PL/SQL)      VARIABLE
CLEAR           HOST            RUN                          WHENEVER OSERROR
COLUMN          INPUT           SAVE                         WHENEVER SQLERROR
COMPUTE         LIST            SET                          XQUERY
CONNECT         PASSWORD        SHOW
```

The `set` command is the primary command used for customizing your environment settings. Listing 1-6 shows the help text for the `set` command.

Listing 1-6. *SQL*Plus SET Command*

```
SQL> help set

 SET
 ---

 Sets a system variable to alter the SQL*Plus environment settings
 for your current session. For example, to:
     -   set the display width for data
     -   customize HTML formatting
     -   enable or disable printing of column headings
     -   set the number of lines per page

 SET system_variable value
```

where system_variable and value represent one of the following clauses:

```
APPI[NFO]{OFF|ON|text}
ARRAY[SIZE] {15|n}
AUTO[COMMIT] {OFF|ON|IMM[EDIATE]|n}
AUTOP[RINT] {OFF|ON}
AUTORECOVERY {OFF|ON}
AUTOT[RACE] {OFF|ON|TRACE[ONLY]}
   [EXP[LAIN]] [STAT[ISTICS]]
BLO[CKTERMINATOR] {.|c|ON|OFF}
CMDS[EP] {;|c|OFF|ON}
COLSEP {_|text}
CON[CAT] {.|c|ON|OFF}
COPYC[OMMIT] {0|n}
COPYTYPECHECK {ON|OFF}
DEF[INE] {&|c|ON|OFF}
DESCRIBE [DEPTH {1|n|ALL}]
   [LINENUM {OFF|ON}] [INDENT {OFF|ON}]
ECHO {OFF|ON}
EDITF[ILE] file_name[.ext]
EMB[EDDED] {OFF|ON}
ERRORL[OGGING] {ON|OFF}
   [TABLE [schema.]tablename]
   [TRUNCATE] [IDENTIFIER identifier]

ESC[APE] {\|c|OFF|ON}
ESCCHAR {@|?|%|$|OFF}
EXITC[OMMIT] {ON|OFF}
FEED[BACK] {6|n|ON|OFF}
FLAGGER {OFF|ENTRY|INTERMED[IATE]|FULL}
FLU[SH] {ON|OFF}
HEA[DING] {ON|OFF}
HEADS[EP] {||c|ON|OFF}
INSTANCE [instance_path|LOCAL]
LIN[ESIZE] {80|n}
LOBOF[FSET] {1|n}
LOGSOURCE [pathname]
LONG {80|n}
LONGC[HUNKSIZE] {80|n}
MARK[UP] HTML [OFF|ON]
   [HEAD text] [BODY text] [TABLE text]
   [ENTMAP {ON|OFF}]
   [SPOOL {OFF|ON}]
   [PRE[FORMAT] {OFF|ON}]
NEWP[AGE] {1|n|NONE}
        NULL text
NUMF[ORMAT] format
NUM[WIDTH] {10|n}
PAGES[IZE] {14|n}
        PAU[SE] {OFF|ON|text}
RECSEP {WR[APPED]|EA[CH]|OFF}
RECSEPCHAR {_|c}
SERVEROUT[PUT] {ON|OFF}
        [SIZE {n | UNLIMITED}]
        [FOR[MAT]  {WRA[PPED] |
                WOR[D_WRAPPED] |
        TRU[NCATED]}]
SHIFT[INOUT] {VIS[IBLE] |
                        INV[ISIBLE]}
SHOW[MODE] {OFF|ON}
        SQLBL[ANKLINES] {OFF|ON}
SQLC[ASE] {MIX[ED] |
        LO[WER] | UP[PER]}
SQLCO[NTINUE] {> | text}
SQLN[UMBER] {ON|OFF}
SQLPLUSCOMPAT[IBILITY]
                {x.y[.z]}
SQLPRE[FIX] {#|c}
SQLP[ROMPT] {SQL>|text}
SQLT[ERMINATOR] {;|c|ON|OFF}
SUF[FIX] {SQL|text}
TAB {ON|OFF}
        TERM[OUT] {ON|OFF}
        TI[ME] {OFF|ON}
TIMI[NG] {OFF|ON}
TRIM[OUT] {ON|OFF}
        TRIMS[POOL] {OFF|ON}
        UND[ERLINE] {-|c|ON|OFF}
VER[IFY] {ON|OFF}
                WRA[P] {ON|OFF}
XQUERY {BASEURI text|
        ORDERING{UNORDERED|
ORDERED|DEFAULT}|
        NODE{BYVALUE|BYREFERENCE|
        DEFAULT}|
        CONTEXT text}
SQL>
```

Given the number of commands available, you can easily customize your environment to best suit you. One thing to keep in mind is that the set commands aren't retained by SQL*Plus when you exit/close the tool. Instead of typing in each of the set commands you want to apply each time you use SQL*Plus, you can create a file named `login.sql`. There are actually two files which SQL*Plus reads by default each time you start it. The first is `glogin.sql` and it can be found in the directory `$ORACLE_HOME/sqlplus/admin`. If this file is found, it is read and the statements it contains are executed. This will allow you to store the SQL*Plus commands and SQL statements that customize your experience across SQL*Plus sessions.

After reading `glogin.sql`, SQL*Plus looks for the `login.sql` file. This file must exist in either the directory from which SQL*Plus was started or in a directory included in the path the environment variable SQLPATH points to. Any commands in `login.sql` will take precedence over those in `glogin.sql`. Since 10g, Oracle reads both `glogin.sql` and `login.sql` each time you either start SQL*Plus or execute the `connect` command from within SQL*Plus. Prior to 10g, the `login.sql` script was only executed when SQL*Plus started. The contents of a common `login.sql` file are shown in Listing 1-7.

Listing 1-7. *A Common login.sql File*

```
SET LINES 3000
Sets width of display line (default 80 characters)
SET PAGES 1000
Sets number of lines per page (default 14 lines)
SET TIMING ON
Sets display of elapsed time (default OFF)
SET NULL <null>
Sets display of nulls to show <null> (default empty)
SET SQLPROMPT '&_user@&_connect_identifier> '
Sets the prompt to show connected user and instance
```

Note the use of the variables `_user` and `_connect_identifier` in the `SET SQLPROMPT` command. These are two examples of predefined variables. You may use any of the following predefined variables in your `login.sql` file or in any other script file you may create:

- _connect_identifier
- _date
- _editor (This variable specifies the editor which is started when you use the edit command.)
- _o_version
- _o_release
- _privilege
- _sqlplus_release
- _user

Executing Commands

There are two types of commands that can be executed within SQL*Plus: SQL statements and SQL*Plus commands. The SQL*Plus commands shown in Listing 1-5 and 1-6 are specific to SQL*Plus and can be used for customizing the environment and executing commands that are specific to SQL*Plus, like

DESCRIBE and CONNECT. Executing a SQL*Plus command requires only that you type the command at the prompt and hit Enter. The command is automatically executed. On the other hand, in order to execute SQL statements, you must use a special character to indicate you wish to execute the entered command. You may use either a semi-colon (;) or a forward slash (/). A semi-colon may be placed directly at the end of the typed command or on a following blank line. The forward slash must be placed on a blank line in order to be recognized. Listing 1-8 shows how these two execution characters are used.

Listing 1-8. *Execution Character Usage*

```
SQL>select empno, deptno from scott.emp where ename = 'SMITH' ;
     EMPNO     DEPTNO
---------- ----------
      7369         20
SQL>select empno, deptno from scott.emp where ename = 'SMITH'
  2  ;
     EMPNO     DEPTNO
---------- ----------
      7369         20
SQL>select empno, deptno from scott.emp where ename = 'SMITH'
  2  /
     EMPNO     DEPTNO
---------- ----------
      7369         20
SQL>select empno, deptno from scott.emp where ename = 'SMITH'
  2
SQL>/
     EMPNO     DEPTNO
---------- ----------
      7369         20
SQL>select empno, deptno from scott.emp where ename = 'SMITH'/
  2
SQL>l
  1* select empno, deptno from scott.emp where ename = 'SMITH'/
SQL>/
select empno, deptno from scott.emp where ename = 'SMITH'/
                                                         *
ERROR at line 1:
ORA-00936: missing expression
```

Notice the fifth example that puts the / at the end of the statement. The cursor moves to a new line instead of executing the command immediately. Then, if you press Enter again, the statement is entered into the SQL*Plus buffer but not executed. In order to view the contents of the SQL*Plus buffer, the list command is used (abbreviated to only l). If you then attempt to execute the statement in the buffer using /, which is how the / command is intended to be used, you get an error. That's because you had typed in the / on the end of the SQL statement line originally. The / is not a valid SQL command and thus causes an error when the statement attempts to execute.

Another way to execute commands is to place them in a file. You can produce these files with the text editor of your choice outside of SQL*Plus or you may invoke an editor directly from SQL*Plus using the `EDIT` command. The `EDIT` command will either open a named file or create a file if it doesn't exist. The file must be in the default directory or you must specify the full path. To set the editor to one of your choice, you simply set the predefined `_editor` variable using the following command: `define _editor='/<full path>/myeditor.exe'`. Files with the extension of `.sql` will execute without having to include the extension and can be ran using either the `@` or `START` command. Listing 1-9 shows the use of both commands.

Listing 1-9. *Executing .sql Script Files*

```
SQL> @list_depts
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
SQL>
SQL> start list_depts
DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
SQL>
SQL>l
  1* select * from scott.dept
SQL>
```

SQL*Plus has many features and options—way too many to cover here. For what we'll need in this book, this brief overview should suffice. However, the Oracle documentation provides guides for SQL*Plus usage and there are numerous books, including *Beginning Oracle SQL*, that go into more depth if you're interested.

The Five Core SQL Statements

The SQL language contains many different statements. In your professional career you may end up using just a small percentage of what is available to you. But isn't that the case with almost any product you use? I once heard a statistic quoted stating that most people use 20 percent or less of the functionality available in the software products or programming languages they regularly use. I don't know if that's actually true or not, but in my experience, it seems fairly accurate. I have found the same basic SQL statement formats in use within most applications for almost 20 years. Very few people ever use everything SQL has to offer—and often improperly implement those they do use frequently. Obviously, we will not be able to cover all the statements and their options found in the SQL language. This book is intended to provide you deeper insight into the most commonly used statements and help you learn to apply them more effectively.

In this book, we will examine five of the most frequently used SQL statements. These statements are SELECT, INSERT, UPDATE, DELETE, and MERGE. Although we'll address each of these core statements in some fashion, the focus will be primarily on the SELECT statement. Developing a good command of these five statements will provide a strong foundation for your day-to-day work with the SQL language.

The SELECT Statement

The SELECT statement is used to retrieve data from one or more tables or other database objects. You should already be familiar with the basics of the SELECT statement so instead of reviewing the statement from that beginner point of view, I wanted to review how a SELECT statement processes logically. You should have already learned the basic clauses that form a common SELECT statement, but in order to build the foundation mindset you'll need to write well-formed and efficient SQL consistently, you need to understand how SQL processes.

How a query statement is processed logically may be quite different from its actual physical processing. The Oracle cost-based optimizer (CBO) is responsible for generating the actual execution plan for a query and we will cover what the optimizer does, how it does it, and why in the chapters ahead. For now, note that the optimizer will determine how to access tables and in which order to process them, and how to join multiple tables and apply filters. The logical order of query processing occurs in a very specific order. However, the steps the optimizer chooses for the physical execution plan can end up actually processing the query in a very different order. Listing 1-10 shows a query stub containing the main clauses of a SELECT statement with step numbers assigned to each clause in the order it is logically processed.

Listing 1-10. *Logical Query Processing Order*

```
5       SELECT  <column list>
1       FROM          <source object list>
1.1     FROM          <left source object> <join type>
                  JOIN <right source object> ON <on predicates>
2       WHERE         <where predicates>
3       GROUP BY      <group by expression(s)>
4       HAVING        <having predicates>
6       ORDER BY      <order by list>
```

You should notice right away that SQL differs from other programming languages in that the first written statement (the SELECT) is not the first line of code that is processed; the FROM clause is processed first. Note that I have shown two different FROM clauses in this listing. The one marked as 1.1 is provided to show the difference when ANSI syntax is used. It may be helpful to imagine that each step in the processing order creates a temporary dataset. As each step is processed, the dataset is manipulated until a final result is formulated. It is this final result set of data that the query returns to the caller.

In order to walk through each part of the SELECT statement in more detail, you'll use the query in Listing 1-11 that returns a result set containing a list of female customers that have placed more than four orders.

Listing 1-11. *Female Customers Who Have Placed More Than Four Orders*

```
SQL> select c.customer_id, count(o.order_id) as orders_ct
  2  from oe.customers c
  3  join oe.orders o
  4  on c.customer_id = o.customer_id
  5  where c.gender = 'F'
  6  group by c.customer_id
  7  having count(o.order_id) > 4
  8  order by orders_ct, c.customer_id
  9  /
CUSTOMER_ID  ORDERS_CT
----------- ----------
        146          5
        147          5
```

The FROM Clause

The FROM clause lists the source objects from which data is selected. This clause can contain tables, views, materialized views, partitions or subpartitions, or may specify a subquery that identifies objects. If multiple source objects are used, this logical processing phase also applies each join type and ON predicates (shown as step 1.1). You'll cover join types in more detail later but note that as joins are processed, they occur in the following order:

1. Cross join, also called a Cartesian product
2. Inner join
3. Outer join

In the example query in Listing 1-11, the FROM clause lists two tables: customers and orders. They are joined on the customer_id column. So, when this information is processed, the initial dataset that will be produced by the FROM clause will include rows where the customer_id matches in both tables. The result set will contain 105 rows at this point. To verify this is true, simply execute only the first four lines of the example query as shown in Listing 1-12.

Listing 1-12. *Partial Query Execution Through the FROM Clause Only*

```
SQL> select c.customer_id cust_id, o.order_id ord_id, c.gender
  2  from oe.customers c
  3  join oe.orders o
  4  on c.customer_id = o.customer_id;

CUST_ID ORD_ID G  CUST_ID ORD_ID G  CUST_ID ORD_ID G
------- ------ -  ------- ------ -  ------- ------ -
    147   2450 F      101   2430 M      109   2394 M
    147   2425 F      101   2413 M      116   2453 M
    147   2385 F      101   2447 M      116   2428 M
    147   2366 F      101   2458 M      116   2369 M
    147   2396 F      102   2431 M      116   2436 M
```

```
    148   2451 M       102   2414 M       117   2456 M
    148   2426 M       102   2432 M       117   2429 M
    148   2386 M       102   2397 M       117   2370 M
    148   2367 M       103   2437 F       117   2446 M
    148   2406 M       103   2415 F       118   2457 M
    149   2452 M       103   2433 F       118   2371 M
    149   2427 M       103   2454 F       120   2373 M
    149   2387 M       104   2438 F       121   2374 M
    149   2368 M       104   2416 F       122   2375 M
    149   2434 M       104   2355 F       123   2376 F
    150   2388 M       104   2354 F       141   2377 M
    151   2389 M       105   2439 F       143   2380 M
    152   2390 M       105   2417 F       144   2445 M
    153   2391 M       105   2356 F       144   2422 M
    154   2392 F       105   2358 F       144   2382 M
    155   2393 M       106   2441 M       144   2363 M
    156   2395 F       106   2418 M       144   2435 M
    157   2398 M       106   2359 M       145   2448 M
    158   2399 M       106   2381 M       145   2423 M
    159   2400 M       107   2442 F       145   2383 M
    160   2401 M       107   2419 F       145   2364 M
    161   2402 M       107   2360 F       145   2455 M
    162   2403 M       107   2440 F       119   2372 M
    163   2404 M       108   2443 M       142   2378 M
    164   2405 M       108   2420 M       146   2449 F
    165   2407 M       108   2361 M       146   2424 F
    166   2408 F       108   2357 M       146   2384 F
    167   2409 M       109   2444 M       146   2365 F
    169   2411 F       109   2421 M       146   2379 F
    170   2412 M       109   2362 M       168   2410 M
105 rows selected.
```

■ **NOTE** I formatted the result of this output manually to make it fit nicely on the page. The actual output was displayed over 105 separate lines.

The WHERE Clause

The WHERE clause provides a way to conditionally limit the rows emitted to the query's final result set. Each condition, or predicate, is entered as a comparison of two values or expressions. The comparison will match (evaluate to TRUE) or it will not match (evaluate to FALSE). If the comparison is FALSE, then the row will not be included in the final result set.

I need to digress just a bit to cover an important aspect of SQL related to this step. Actually, the possible values of a logical comparison in SQL are TRUE, FALSE, and UNKNOWN. The UNKNOWN value occurs when a null is involved. Nulls compared to anything or nulls used in expressions evaluate to null, or UNKNOWN. A null represents a missing value and can be confusing due to inconsistencies in how nulls are treated within different elements of the SQL language. We'll address how nulls effect the execution of SQL statements throughout the book, but I didn't want to ignore mentioning the topic at this point. What I stated previously is still basically true, that comparisons will either return TRUE or FALSE. What you'll find is that when a null is involved in a filter comparison, it is treated as if it were FALSE.

In our example, there is a single predicate used to limit the result to only females who have placed orders. If you review the intermediate result after the FROM clause was processed (see Listing 1-12), you'll note that only 31 of the 105 rows were placed by female customers (gender = 'F'). Therefore, after the WHERE clause is applied, the intermediate result set would be reduced from 105 down to 31 rows.

After the WHERE clause is applied, the detailed result set is ready. Note that I use the phrase "detailed result set." What I mean is the rows that satisfy your query requirements are now available. Other clauses may be applied (GROUP BY, HAVING) that will aggregate and further limit the final result set that the caller will receive, but it is important to note that at this point, all the data your query needs to compute the final answer is available.

The WHERE clause is intended to restrict, or reduce, the result set. The less restrictions you include, the more data your final result set will contain. The more data you need to return, the longer the query will take to execute.

The GROUP BY Clause

The GROUP BY clause aggregates the filtered result set available after processing the FROM and WHERE clauses. The selected rows are grouped by the expression(s) listed in this clause to produce a single row of summary information for each group. You may group by any column of any object listed in the FROM clause even if you don't intend to display that column in the list of output columns. Conversely, any non-aggregate column in the select list must be included in the GROUP BY expression.

There are two additional operations that can be included in a GROUP BY clause: ROLLUP and CUBE. The ROLLUP operation is used to produce subtotal values. The CUBE operation is used to produce cross-tabulation values. If you use either of these operations, you'll get more than one row of summary information. Both of these operations will be discussed in detail in Chapter 7: Advanced Grouping.

In the example query, the requested grouping is by customer_id. This means that there will only be one row for each distinct customer_id. Of the 31 rows that represent the females who have placed orders that have made it through the WHERE clause processing, there are 11 distinct customer_id values, as shown in Listing 1-13.

Listing 1-13. *Partial Query Execution Through the GROUP BY Clause*

```
SQL> select c.customer_id, count(o.order_id) as orders_ct
  2  from oe.customers c
  3  join oe.orders o
  4  on c.customer_id = o.customer_id
  5 where gender = 'F'
  6 group by c.customer_id;

CUSTOMER_ID  ORDERS_CT
----------- ----------
        156          1
        123          1
        166          1
```

```
        154          1
        169          1
        105          4
        103          4
        107          4
        104          4
        147          5
        146          5
11 rows selected.
```

You'll notice that the output from the query, while grouped, is not ordered. The display makes it appear as though the rows are ordered by `order_ct`, but this is more coincidence and not guaranteed behavior. This is an important item to remember: the `GROUP BY` clause does not insure ordering of data. If you want the list to display in a specific order, you have to specify an `ORDER BY` clause.

The HAVING Clause

The `HAVING` clause restricts the grouped summary rows to those where the condition(s) in the clause are `TRUE`. Unless you include a `HAVING` clause, all summary rows are returned. The `GROUP BY` and `HAVING` clauses are actually interchangeable positionally; it doesn't matter which one comes first. However, it seems to make more sense to code them with the `GROUP BY` first since `GROUP BY` is logically processed first. Essentially, the `HAVING` clause is a second `WHERE` clause that is evaluated after the `GROUP BY` occurs and is used to filter on grouped values.

In our example query, the `HAVING` clause, `HAVING COUNT(o.order_id) > 4`, limits the grouped result data of 11 rows down to 2. You can confirm this by reviewing the list of rows returned after the `GROUP BY` is applied, as shown in Listing 1-13. Note that only customers 146 and 147 have placed more than four orders. The two rows that make up the final result set are now ready.

The SELECT List

The `SELECT` list is where the columns included in the final result set from your query are provided. A column can be an actual column from a table, an expression, or even the result of a `SELECT` statement, as shown in Listing 1-14.

Listing 1-14. *Example Query Showing Select List Alternatives*

```
SQL> select c.customer_id, c.cust_first_name||' '||c.cust_last_name,
  2  (select e.last_name from hr.employees e where e.employee_id = c.account_mgr_id)
acct_mgr)
  3  from oe.customers c;


    CUSTOMER_ID CUST_NAME                                  ACCT_MGR
--------------- ---------------------------------------- --------------
            147 Ishwarya Roberts                          Russell
            148 Gustav Steenburgen                        Russell
...
```

```
          931 Buster Edwards                          Cambrault
          981 Daniel Gueney                           Cambrault
319 rows selected.
```

When another SELECT statement is used to produce the value of a column, the query must return only one row and one column value. These types of subqueries are referred to as scalar subqueries. While this can be very useful syntax, keep in mind that the scalar subquery will be executed once for each row in the result set. There are optimizations available that may eliminate some duplicate executions of the subquery, but the worse case scenario is that each row will require this scalar subquery to be executed. Imagine the possible overhead involved if your result set had thousands, or millions, of rows! We'll review scalar subqueries later in the book and discuss how to use them optimally.

Another option you may need to use in the SELECT list is the DISTINCT clause. The example doesn't use it, but I wanted to mention it briefly. The DISTINCT clause causes duplicate rows to be removed from the data set produced after the other clauses have been processed.

After the select list is processed, you now have the final result set for your query. The only thing that remains to be done, if it is included, is to sort the result set into a desired order.

The ORDER BY Clause

The ORDER BY clause is used to order the final set of rows returned by the statement. In this case, the requested sort order was to be by orders_ct and customer_id. The orders_ct column is the value computed using the COUNT aggregate function in the GROUP BY clause. As shown in Listing 1-13, there were two customers that each placed more than four orders. Since each customer placed five orders, the order_ct is the same, so the second ordering column determines the final display order. As shown in Listing 1-15, the final sorted output of the query is a two row data set ordered by customer_id.

Listing 1-15. *Example Query Final Output*

```
SQL> select c.customer_id, count(o.order_id) as orders_ct
  2  from oe.customers c
  3  join oe.orders o
  4  on c.customer_id = o.customer_id
  5  where c.gender = 'F'
  6  group by c.customer_id
  7  having count(o.order_id) > 4
  8  order by orders_ct, c.customer_id
  9  /
CUSTOMER_ID  ORDERS_CT
----------- ----------
        146          5
        147          5
```

When ordered output is requested, Oracle must take the final set of data after all other clauses have been processed and sort them as specified. The size of the data that needs to be sorted is important. When I say size, I mean total bytes of data that is in the result set. To estimate the size of the data set, you multiply the number of rows by the number of bytes per row. The bytes per row are determined by summing the average column lengths of each of the columns in the select list.

The example query requests only the `customer_id` and `orders_ct` column values in the select list. Let's use 10 as our estimated bytes per row value. I'll show you in Chapter 6 where to find the optimizer's estimate for this value. So, given that we only have two rows in the result set, the sort size is actually quite small, approximately 20 bytes. Remember that this is only an estimate, but the estimate is an important one.

Small sorts should be accomplished entirely in memory while large sorts may have to use temporary disk space to complete the sort. As you can likely deduce, a sort that occurs in memory will be faster than a sort that must use disk. Therefore, when the optimizer estimates the effect of sorting data, it has to consider how big the sort is in order to adjust how to accomplish getting the query result in the most efficient way. In general, consider sorts as a fairly expensive overhead to your query processing time, particularly if the size of your result set is large.

The INSERT Statement

The `INSERT` statement is used to add rows to a table, partition, or view. Rows can be inserted in either a single-table or multi-table method. A single-table insert will insert values into one row of one table by either explicitly specifying the values or by retrieving the values using a subquery. The multi-table insert will insert rows into one or more tables and will compute the row values it inserts by retrieving the values using a subquery.

Single-table Inserts

The first example in Listing 1-16 illustrates a single-table insert using the values clause. Each column value is explicitly entered. The column list is optional if you include values for each column defined in the table. However, if you only want to provide values for a subset of the columns, you must specify the column names in the column list. A good practice is to include the column list regardless of whether or not you specify values for all the columns. Doing so acts to self-document the statement and also can help reduce possible errors that might happen in the future should someone add a new column to the table.

Listing 1-16. *Single-Table Insert*

```
SQL> insert into hr.jobs (job_id, job_title, min_salary, max_salary)
  2  values ('IT_PM', 'Project Manager', 5000, 11000) ;

1 row created.

SQL> insert into scott.bonus (ename, job, sal)
  2  select ename, job, sal * .10
  3  from scott.emp;

14 rows created.
```

The second example illustrates an insert using a subquery. This is a very flexible option for inserting rows. The subquery can be written to return one or more rows. Each row returned will be used to supply column values for the new rows to be inserted. The subquery can be as simple or complex as needed to satisfy your needs. In this example, we use the subquery to compute a 10% bonus for each employee based on their current salary. The bonus table actually has four columns, but we only populate three of them with this insert. The `comm` column isn't populated with a value from the subquery and we do not include it in the column list. Since we don't include this column, the value for that column will be null. Note that if the comm column had a not null constraint, you would have gotten a constraint error and the statement would have failed.

Multi-table Inserts

The multi-table insert example in Listing 1-17 illustrates how rows returned from a single subquery can be used to insert rows into more than one table. We start with three tables: `small_customers`, `medium_customers`, and `large_customers`. We'd like to populate these tables with customer data based on the total amount of orders a customer has placed. The subquery sums the `order_total` column for each customer and then the insert conditionally places a row in the proper table based on whether the customer is considered to be small (less than $10,000 of total orders), medium (between $10,000 and $99,999.99), and large (greater than or equal to $100,000).

Listing 1-17. *Multi-Table Insert*

```
SQL> select * from small_customers ;

no rows selected

SQL> select * from medium_customers ;

no rows selected

SQL> select * from large_customers ;

no rows selected

SQL> insert all
     when sum_orders < 10000 then
     into small_customers
     when sum_orders >= 10000 and sum_orders < 100000 then
     into medium_customers
     else
     into large_customers
     select customer_id, sum(order_total) sum_orders
     from oe.orders
     group by customer_id ;

47 rows created.

SQL> select * from small_customers ;
```

```
CUSTOMER_ID SUM_ORDERS
----------- ----------
        120        416
        121       4797
        152     7616.8
        157     7110.3
        160      969.2
        161        600
        162        220
        163        510
        164       1233
        165       2519
        166        309
        167         48

12 rows selected.

SQL> select * from medium_customers ;

CUSTOMER_ID SUM_ORDERS
----------- ----------
        102    69211.4
        103    20591.4
        105    61376.5
        106    36199.5
        116      32307
        119    16447.2
        123    11006.2
        141    38017.8
        142    25691.3
        143    27132.6
        145    71717.9
        146    88462.6
        151      17620
        153    48070.6
        154      26632
        155    23431.9
        156      68501
        158    25270.3
        159    69286.4
        168      45175
        169    15760.5
        170      66816

22 rows selected.
```

```
SQL> select * from large_customers ;
CUSTOMER_ID SUM_ORDERS
----------- ----------
        101   190395.1
        104   146605.5
        107   155613.2
        108   213399.7
        109   265255.6
        117   157808.7
        118   100991.8
        122   103834.4
        144   160284.6
        147   371278.2
        148   185700.5
        149   403119.7
        150   282694.3

13 rows selected.
```

Note the use of the `ALL` clause following the `INSERT` keyword. When `ALL` is specified, the statement will perform unconditional multi-table inserts. This means that each `WHEN` clause is evaluated for each row returned by the subquery regardless of the outcome of a previous condition. Therefore, you need to be careful about how you specify each condition. For example, if I had used `WHEN sum_orders < 100000` instead of the range I specified, the `medium_customers` table would have included the rows that were also inserted into `small_customers`.

You should specify the `FIRST` option to cause each `WHEN` to be evaluated in the order it appears in the statement and to skip subsequent `WHEN` clause evaluations for a given subquery row. The key is to remember which option, `ALL` or `FIRST`, best meets your needs and then use the one most suitable.

The UPDATE Statement

The `UPDATE` statement is used to change the column values of existing rows in a table. The syntax for this statement is composed of three parts: `UPDATE`, `SET`, and `WHERE`. The `UPDATE` clause specifies the table to update. The `SET` clause specifies which columns are changed and the modified values. The `WHERE` clause is used to conditionally filter which rows will be updated. It is optional and if it is omitted, the update operation will be applied to all rows of the specified table.

Listing 1-18 demonstrates several different ways an `UPDATE` statement can be written. First, I create a duplicate of the `employees` table called `employees2`, then I execute several different updates that accomplish basically the same task: the employees in department 90 are updated to have a 10% salary increase and, in the case of Example 5, the `commission_pct` column is also updated. Following are the different approaches taken:

Example 1: Update a single column value using an expression.

Example 2: Update a single column value using a subquery.

Example 3: Update single column using subquery in `WHERE` clause to determine which rows to update.

Example 4: Update a table using a `SELECT` statement to define the table and column values.

Example 5: Update multiple columns using a subquery.

Listing 1-18. *UPDATE Statement Examples*

```
SQL> -- create a duplicate employees table
SQL> create table employees2 as select * from employees ;
Table created.

SQL> -- add a primary key
SQL> alter table employees2
  1  add constraint emp2_emp_id_pk primary key (employee_id) ;

Table altered.

SQL> -- retrieve list of employees in department 90
SQL> select employee_id, last_name, salary
  2  from employees where department_id = 90 ;

    EMPLOYEE_ID LAST_NAME                          SALARY
--------------- ------------------------- ---------------
            100 King                                24000
            101 Kochhar                             17000
            102 De Haan                             17000

3 rows selected.

SQL> -- Example 1: Update a single column value using an expression

SQL> update employees2
  2  set salary = salary * 1.10 -- increase salary by 10%
  3  where department_id = 90 ;

3 rows updated.

SQL> commit ;

Commit complete.
```

```
SQL> select employee_id, last_name, salary
  2  from employees2 where department_id = 90 ;

EMPLOYEE_ID LAST_NAME  SALARY
----------- ---------- ------
        100 King        26400   -- previous value 24000
        101 Kochhar     18700   -- previous value 17000
        102 De Haan     18700   -- previous value 17000

3 rows selected.

SQL> -- Example 2: Update a single column value using a subquery

SQL> update employees
  2  set salary = (select employees2.salary
  3                 from employees2
  4                where employees2.employee_id = employees.employee_id
  5                  and employees.salary != employees2.salary)
  6  where department_id = 90 ;

3 rows updated.

SQL> select employee_id, last_name, salary
  2  from employees where department_id = 90 ;

    EMPLOYEE_ID LAST_NAME                          SALARY
--------------- ------------------------- ---------------
            100 King                                26400
            101 Kochhar                             18700
            102 De Haan                             18700

3 rows selected.

SQL> rollback ;

Rollback complete.

SQL> -- Example 3: Update single column using subquery in
SQL> -- WHERE clause to determine which rows to update

SQL> update employees
  2  set salary = salary * 1.10
  3  where department_id in (select department_id
  4                           from departments
  5                          where department_name = 'Executive') ;

3 rows updated.
```

```
SQL> select employee_id, last_name, salary
  2  from employees
  3  where department_id in (select department_id
  4                           from departments
  5                          where department_name = 'Executive') ;

    EMPLOYEE_ID LAST_NAME                          SALARY
--------------- ------------------------- ---------------
            100 King                                26400
            101 Kochhar                             18700
            102 De Haan                             18700

3 rows selected.

SQL> rollback ;

Rollback complete.

SQL> -- Example 4: Update a table using a SELECT statement
SQL> -- to define the table and column values

SQL> update (select e1.salary, e2.salary new_sal
  2            from employees e1, employees2 e2
  3           where e1.employee_id = e2.employee_id
  4             and e1.department_id = 90)
  5  set salary = new_sal;

3 rows updated.

SQL> select employee_id, last_name, salary, commission_pct
  2  from employees where department_id = 90 ;

    EMPLOYEE_ID LAST_NAME                          SALARY  COMMISSION_PCT
--------------- ------------------------- --------------- ---------------
            100 King                                26400
            101 Kochhar                             18700
            102 De Haan                             18700

3 rows selected.

SQL> rollback ;

Rollback complete.

SQL> -- Example 5: Update multiple columns using a subquery
```

```
SQL> update employees
  2  set (salary, commission_pct) = (select employees2.salary, .10 comm_pct
  3                 from employees2
  4                where employees2.employee_id = employees.employee_id
  5                  and employees.salary != employees2.salary)
  6  where department_id = 90 ;

3 rows updated.
SQL> select employee_id, last_name, salary, commission_pct
  2  from employees where department_id = 90 ;

    EMPLOYEE_ID LAST_NAME                          SALARY  COMMISSION_PCT
--------------- ------------------------- -------------- ---------------
            100 King                                26400              .1
            101 Kochhar                             18700              .1
            102 De Haan                             18700              .1

3 rows selected.

SQL> rollback ;

Rollback complete.

SQL>
```

The DELETE Statement

The DELETE statement is used to remove rows from a table. The syntax for this statement is composed of three parts: DELETE, FROM, and WHERE. The DELETE keyword stands alone. Unless you decide to use a hint, which we'll discuss later, there are no other options associated with the DELETE keyword. The FROM clause identifies the table from which rows will be deleted. As the examples in Listing 1-19 demonstrate, the table can be specified directly or via a subquery. The WHERE clause provides any filter conditions to help determine which rows are deleted. If the WHERE clause is omitted, the delete operation will delete all rows in the specified table.

Listing 1-19 demonstrates several different ways a DELETE statement can be written. Note that I am using the employees2 table created in Listing 1-18 for these examples. Here are the different delete methods that you'll see:

Example 1: Delete rows from specified table using a filter condition in the WHERE clause.

Example 2: Delete rows using a subquery in the FROM clause.

Example 3: Delete rows from specified table using a subquery in the WHERE clause.

Listing 1-19. *DELETE Statement Examples*

```
SQL> select employee_id, department_id, last_name, salary
  2  from employees2
  3  where department_id = 90;
```

```
    EMPLOYEE_ID   DEPARTMENT_ID LAST_NAME                                 SALARY
--------------- --------------- ------------------------- ---------------
            100              90 King                                   26400
            101              90 Kochhar                                18700
            102              90 De Haan                                18700

3 rows selected.
SQL> -- Example 1: Delete rows from specified table using
SQL> -- a filter condition in the WHERE clause
SQL> delete from employees2
  2  where department_id = 90;

3 rows deleted.

SQL> select employee_id, department_id, last_name, salary
  2  from employees2
  3  where department_id = 90;

no rows selected

SQL> rollback;

Rollback complete.

SQL> select employee_id, department_id, last_name, salary
  2  from employees2
  3  where department_id = 90;

    EMPLOYEE_ID   DEPARTMENT_ID LAST_NAME                                 SALARY
--------------- --------------- ------------------------- ---------------
            100              90 King                                   26400
            101              90 Kochhar                                18700
            102              90 De Haan                                18700

3 rows selected.

SQL> -- Example 2: Delete rows using a subquery in the FROM clause
SQL> delete from (select * from employees2 where department_id = 90);

3 rows deleted.

SQL> select employee_id, department_id, last_name, salary
  2  from employees2
  3  where department_id = 90;

no rows selected
```

```
SQL> rollback;

Rollback complete.

SQL> select employee_id, department_id, last_name, salary
  2  from employees2
  3  where department_id = 90;
    EMPLOYEE_ID   DEPARTMENT_ID LAST_NAME                          SALARY
--------------- --------------- ------------------------- ---------------
            100              90 King                                26400
            101              90 Kochhar                             18700
            102              90 De Haan                             18700

3 rows selected.

SQL> -- Example 3: Delete rows from specified table using
SQL> -- a subquery in the WHERE clause
SQL> delete from employees2
  2  where department_id in (select department_id
  3                      from departments
  4                     where department_name = 'Executive');

3 rows deleted.

SQL> select employee_id, department_id, last_name, salary
  2  from employees2
  3  where department_id = 90;

no rows selected

SQL> rollback;

Rollback complete.

SQL>
```

The MERGE Statement

The MERGE statement is a single command that combines the ability to update or insert rows into a table by conditionally deriving the rows to be updated or inserted from one or more sources. It is most frequently used in data warehouses to move large amounts of data but its use is not limited to only data warehouse environments. The big value-add this statement provides is that you have a convenient way to combine multiple operations into one. This allows you to avoid issuing multiple INSERT, UPDATE, and DELETE statements. And, as you'll see later in the book, if you can avoid doing work you really don't have to do, your response times will likely improve.

The syntax for the MERGE statement is:

```
MERGE <hint>
INTO <table_name>
USING <table_view_or_query>
ON (<condition>)
WHEN MATCHED THEN <update_clause>
DELETE <where_clause>
WHEN NOT MATCHED THEN <insert_clause>
[LOG ERRORS <log_errors_clause> <reject limit <integer | unlimited>];
```

In order to demonstrate the use of the MERGE statement, Listing 1-20 shows how to create a test table and then appropriately insert or update rows into that table based on the MERGE conditions.

Listing 1-20. *MERGE Statement Example*

```
SQL> create table dept60_bonuses
  2  (employee_id       number
  3  ,bonus_amt         number);

Table created.

SQL> insert into dept60_bonuses values (103, 0);

1 row created.

SQL> insert into dept60_bonuses values (104, 100);

1 row created.

SQL> insert into dept60_bonuses values (105, 0);

1 row created.

SQL> commit;

Commit complete.

SQL> select employee_id, last_name, salary
  2  from employees
  3  where department_id = 60 ;

    EMPLOYEE_ID LAST_NAME                          SALARY
--------------- ------------------------- ---------------
            103 Hunold                               9000
            104 Ernst                                6000
            105 Austin                               4800
```

```
         106 Pataballa                                 4800
         107 Lorentz                                   4200

5 rows selected.

SQL> select * from dept60_bonuses;
   EMPLOYEE_ID       BONUS_AMT
--------------- ---------------
            103               0
            104             100
            105               0

3 rows selected.

SQL> merge into dept60_bonuses b
     using (
       select employee_id, salary, department_id
       from employees
       where department_id = 60) e
     on (b.employee_id = e.employee_id)
     when matched then
       update set b.bonus_amt = e.salary * 0.2
        where b.bonus_amt = 0
       delete where (e.salary > 7500)
     when not matched then
       insert (b.employee_id, b.bonus_amt)
       values (e.employee_id, e.salary * 0.1)
       where (e.salary < 7500);

4 rows merged.

SQL> select * from dept60_bonuses;

   EMPLOYEE_ID       BONUS_AMT
--------------- ---------------
            104             100
            105             960
            106             480
            107             420

4 rows selected.

SQL> rollback;

Rollback complete.

SQL>
```

The MERGE accomplished the following:

- Two rows were inserted (employee_ids 106 and 107).
- One row was updated (employee_id 105).
- One row was deleted (employee_id 103).
- One row remained unchanged (employee_id 104).

Without the MERGE statement, you would have had to write at least three different statements to complete the same work.

Summary

As you can tell from the examples shown so far, the SQL language offers many alternatives that can produce the same result set. What you may have also noticed is that each of the five core statements can utilize similar constructs, like subqueries. The key is to learn which constructs are the most efficient under various circumstances. We'll cover how to do that later in the book.

If you had any trouble following the examples in this chapter, make sure to take the time to review either *Beginning Oracle SQL* or the *SQL Reference Guide* in the Oracle documentation. The rest of the book will assume you are comfortable with the basic constructs for each of the five core SQL statements: SELECT, INSERT, UPDATE, DELETE, and MERGE.