1

Exploring Popular SQL Implementations

Any tour into the realm of writing SQL functions should begin with a solid foundation of the basic principles of SQL. In this chapter, we will be discussing the ins and outs of creating, querying, and modifying databases using basic SQL syntax. This chapter is the basis upon which we will build in the following chapters. This will help you unravel the mystery of using the power of built-in functions available across the various relational database management systems (RDBMS) platforms and to introduce new functionality into your applications by developing your own user-defined functions (UDFs).

Introduction to SQL

Structured Query Language (otherwise known as SQL and pronounced "SEE-kwul") was first developed by IBM in the mid- to late 1970s for their DB2 platform RDBMS. At the time, its purpose was to provide a way in which the RDBMS could retrieve data in a declarative way. Declarative "programming" was a way in which the RDBMS developer could specify what data would be selected, inserted, updated, or deleted without having to necessarily know where the data was or how it was stored. That was the job of the RDBMS. The main goal of SQL was to provide the following functionality to the RDBMS:

- Query the database to retrieve the data stored therein.
- Update existing data within the database.
- Insert new data into the database.
- Remove unwanted data from the database.
- Add permissions to RDBMS objects (databases, tables, and so on).
- Modify a database's structure.
- Change security settings.

Oracle released the first commercial RDBMS that used SQL in 1978. Soon after that, in the mid-1980s, Sybase released its own RDBMS, SQL Server. In 1988, this was ported to OS/2 by Sybase and Microsoft, and eventually to Windows NT. In 1993, the partnership parted ways and Sybase eventually renamed their product *Adaptive Server Enterprise (ASE)* to differentiate it from the Microsoft version. However, since that time, Open Source RDBMS solutions such as MySQL and PostgreSQL have taken hold in the marketplace and are among the fastest growing. Even though all of these systems follow or attempt to follow the same base SQL implementation, each has its own unique characteristics and extensions that make it stand out from the rest.

Understanding the SQL Standard

The "standard" of SQL is laid out by both the American National Standards Institute (ANSI) and the International Standards Organization (ISO). It is fundamentally a set of base standards that, in theory, is the agreed-upon example of what the SQL syntax and logic in an RDBMS should be. The original ANSI standard was put forth in 1986 and then later developed into ANSI SQL:1989 or, simply, SQL 89. The former version laid out a basic pattern of the following three separate ways in which SQL could be implemented:

- **Embedded** — This refers to embedding SQL statements within a program separate from the RDBMS instance. Patterns for this implementation were written to reflect the programming languages of the day (COBAL, FORTRAN, Pascal, and PL/1).
- **Direct** — The implementer could provide specific direct implementation of SQL to the developer.
- **Module** — Modules enable calls from procedures within programs and return a value to the calling program.

This baseline was further expanded with the release of ANSI 1992 or SQL 92, which included such things as connections to databases, dynamic SQL, and outer joins. This was in addition to establishing levels of compliance (entry, intermediate, and full) that the RDBMS system could tout to their user base. SQL:1999 again extended the standard by including more data types in the mix, such as arrays, user-defined types (UDTs), Boolean, BLOB, and CLOB. SQL:2003 expands upon the data types available, along with some new built-in functions.

However, it should be noted that the standards for core compliance have not been changed from the SQL:1999 version. This effectively means that anything that is SQL 99-compliant will automatically be SQL:2003-compliant as well. Therefore, it is important to understand that even though a specific RDBMS implementation may by SQL:2003-compliant, it does not mean that it has implemented all of the changes proposed within the new standard. In actuality, it has just prescribed to a set of core compliance standards, and you will have to check the specific instance of your RDBMS documentation to see which parts have truly been implemented.

Overview of Vendor Implementations of SQL

Another important aspect of understanding the basics of ANSI SQL is to know the different implementations of available RDBMS packages. This book is based upon what we see as the top six RDBMS implementations that use the ANSI SQL standard. It is important to realize that even though the ANSI SQL

standard is something that every database should strive to emulate, each vendor implementation of that standard is unique and has nuances that the other implementations will not have. This section provides an overview of each of the six different RDBMS implementations discussed in this book to give you a good understanding of their backgrounds before going into the technical details of each implementation.

Oracle

Oracle is the market leader in the commercial RDBMS market. As mentioned in the previous section, Oracle released the first commercially available RDBMS in 1978. With the release of its commercial version 10g, Oracle claims SQL:2003 compliance, as well as a full set of features and tool sets for the developer. In addition, Oracle provides several different editions that fit a wide variety of operating environments (UNIX, Linux, and Windows) and levels of use.

IBM DB2 UDB

IBM created SQL for their DB2 database platform in the mid- to late 1970s. IBM is the world's leading hardware vendor, and their DB2 database platform has evolved into their Universal Database line. As of this writing, the current version is 8.2, and they offer several different editions to fit a number of operating platforms and uses.

Microsoft SQL Server and Sybase

As mentioned earlier, Sybase produced the original SQL Server and Microsoft later entered into a code-velopment agreement with them. Later in 1988, Microsoft ported the database from OS/2 over to their Windows platform. In 1993, the two companies discontinued their agreement and parted ways.

Since the split in 1993, Sybase has renamed their products *Sybase Adaptive Server Enterprise* and *Sybase IQ*. Originally developed for the OS/2 environment, Sybase's RDMS is now available for a wide variety of platforms, such as Windows, Linux, Solaris x86/64-bit, and Mac.

Microsoft is now the world's largest software developer. SQL Server 2000 is now their flagship database version, and they will soon release SQL Server 2005 (its beta version is currently available). Microsoft touts the ease of use of their database system, rather than total ANSI compliance. They provide a feature-rich set of administration tools, but are limited to the Microsoft Windows platform. SQL Server's most obvious difference from other SQL implementations is its use of Microsoft's extension language T-SQL. In Microsoft's version, there is no apparent difference to distinguish T-SQL from the standard version of SQL: they are treated as one and the same.

MySQL

Possibly the world's most-used Open Source RDBMS, MySQL is the product of its parent company MySQL AB, which was founded in Sweden in the 1980s. Although its solution is marketed as Open Source, it does have a commercial license if developers want to produce closed-source solutions with it. MySQL was initially developed with speed in mind and, as such, has suffered somewhat from its lack of a full feature set compared with some of its competitors. However, in recent releases (such as MySQL 4.0 and 5.0 Alpha), MySQL is starting to tout both its speed and compliance with the new SQL:2003 standard.

PostgreSQL

PostgreSQL (pronounced "Post-gres-Q-L") was developed in 1986 at the University of California at Berkeley. Today, it is considered to be the most powerful of the available Open Source RDBMS packages. Initially, PostgreSQL was written to perform on a number of versions of the UNIX operating system. As of this writing, PostgreSQL is in version 8.0 Beta 4, and is available since the initial release of version 8.0 on Windows NT-based systems. Whereas MySQL was written to be fast, PostgreSQL was developed to be "full featured." However, the trend has shifted with each subsequent release to make it faster.

Connecting to SQL Databases

To connect to an RDBMS, you must have two things: an SQL client and a `CONNECTION` statement. The SQL client acts as (or at least is perceived to be) a part of the specific SQL implementation. In addition, it also helps keep track of the state of all three parts of the connection instance: itself, the SQL agent, and the SQL server. Different vendors will have their own versions (even though the functionality is similar) of the SQL client. Oracle has SQL-PLUS and SQLPlus Worksheet, Microsoft had their own version in Query Analyzer, MySQL has MySQLGUI, and the list goes on. Once you have your particular client that matches your RDBMS of choice, connecting to a database is a rather simple matter. This is done by issuing the following SQL command:

```
CONNECT TO <database_name>
```

If your SQL client is set up with a default database to connect to, then you can simply issue the default connection command, which is the following:

```
 CONNECT TO DEFAULT
```

Please note that if there is no default database established for the SQL client, then an exception error will be thrown by the client because it will not be able to establish a connection. However, depending on which RDBMS system you have and how your security context is set up, after issuing the command, you will more often than not be challenged for a username/password combination to access the database itself.

Once you are connected to your particular database instance, you can change to another database by issuing another `CONNECT` statement, naming the connection, and then issuing the `SET` statement to move between your different named connections, as shown here:

```
CONNECT TO DATABASE1 AS FIRSTDB USER sa;
CONNECT TO DATABASE2 AS SECONDDB USER sa;
SET CONNECTION FIRSTDB;
// Now we are working with the first database... //
SET CONNECTION SECONDDB;
// Now we are working with the second database.... //
```

Once you are finished working within your database(s), it is also necessary to disconnect from your session using the `DISCONNECT` statement, as shown here:

```
DISCONNECT <connection_name>;
// Or if you have multiple connections that you want to close... //
DISCONNECT ALL;
```

ANSI SQL Data Types

The ANSI SQL standard also specifies different data types in which to hold your data. In general, each RDBMS implementation will vary from these as each uses its own unique set to provide some uniqueness to the environment. The basic set of SQL data types can be broken down into the categories shown in the following table.

Type	ANSI SQL Data Types
Boolean	`BOOLEAN`
String	`CHAR` `VARCHAR`

Numeric	`NUMERIC`
	`DECIMAL`
	`DOUBLE PRECISION`
	`FLOAT`
	`INTEGER`
	`REAL`
	`SMALLINT`
	`BIGINT`

Date Time	`DATE`
	`TIME`
	`TIMESTAMP`

Since each RDBMS implementation differs, it is always best to refer to the documentation of your specific vendor to determine which data types it supports. The preceding table should merely serve as a guideline for some of the basic data types that are supported by the ANSI standard. Your particular vendor could have more or less, depending on the implementation. It is equally important to read the vendor descriptions of data types carefully, because, even though two vendors may provide data types with the same name, their precision and range may differ.

Creating SQL Databases

To create a new SQL database, you must issue the following command:

```
CREATE DATABASE <database_name>
```

This is the basic syntax for the `CREATE DATABASE` statement. There are plenty of optional clauses that can be used with the statement, but they differ from implementation to implementation. Clauses include those that specify everything for location of data files, database collation, and database state. It is best to check with your RDBMS systems documentation for the particular variance that they support.

Once the foundation of the database has been created, it is time to focus on creating the tables that will hold the various data you need within the database. The basic syntax for the `CREATE TABLE` statement is as follows:

```
CREATE TABLE <table_name>
(
<column_name1> data_type,
<column_name2> data_type,
.......
)
```

The columns can be identified using any of the ANSI-supported data types detailed earlier in this chapter.

At least one column name of a specific data type must be designated at the time of creation. If all columns are not named when the table is created, then the table may be altered using the `ALTER TABLE` syntax detailed here:

```
ALTER TABLE <table_name>
          [ADD COLUMN <column name> data type]
          [ALTER COLUMN <column name> new data type]
          [DROP COLUMN <column name>]
```

As you can see from this syntax, the `ALTER TABLE` syntax can be used not only to add columns to your table, but also to either modify them or drop them altogether.

Additionally, at most one column of the table can be designated as an identity column. An identity column is automatically assigned values based on an internal sequence generator every time a new row is inserted into the table. These identity columns are important in database table design because they ensure that each row is unique in at least one column. You can implement an identity column in your table by using syntax similar to the following:

```
CREATE TABLE employees
(
EMP_ID INTEGER
                GENERATED ALWAYS AS IDENTITY
                 START WITH 1
                       INCREMENT 1
                       MINVALUE 1
                       NO MAXVALUE
                       NO CYCLE,
EMP_NAME        VARCHAR (30),
EMP_ADDRESS     VARCHAR (50),
EMP_CITY        VARCHAR (20),
EMP_STATE       CHAR (2),
EMP_SALARY      DECIMAL (10, 2)
)
```

A table may also be created with an *index*, which is the equivalent of a table of contents for your table. Indices are created separately from the actual data within the table, and are used to speed up queries on the table. There are several different implementations of placing indices on tables, as shown in the following syntax:

```
CREATE [UNIQUE] INDEX <index name>
     ON <table_name> (column_name1,column_name2,.......)
```

You may optionally use the UNIQUE keyword on your index to specify that any two index values must be unique across your index on the table. Additionally, you may specify more than one column on which to place the index, in which case it is commonly referred to as a *covering index*. It should also be specified that in order to speed up your queries, the index must be placed on a column(s) that your queries are using. In our previous example of creating the employees table, if we place an index on EMP_ID and perform a query on EMP_NAME our index would not be used.

Carefully planning your database creation will keep you from having to spend excess time and resources reconfiguring different aspects of your database structure.

Querying SQL Databases

Querying information from the database involves using the SELECT statement, whose basic syntax is as follows:

```
SELECT select_list
FROM table_source
```

The breakdown of the SELECT statement is similar to any SQL query in that it is composed of keywords and clauses. *Keywords* are the individual SQL statements. In this case, SELECT and FROM would be the keywords. *Clauses* are everything else within the statement and are objects that shape what data the keywords operate upon. In a simple example, we could pull the First_Name and Last_Name from a table named "Authors" in our database, as follows:

```
SELECT FIRST_NAME, LAST_NAME
FROM AUTHORS;
FIRST_NAME            LAST_NAME
Tennessee             Williams
Steven                King
Danielle              Steeley
Margo                 Hennesay
Jordan                Michaels
```

By looking closely at the example, you can see several characteristics of the SELECT statement that are true for all SQL statements in general. The first thing that you should see is that the objects in the select list (FIRST_NAME, LAST_NAME) are pulled from the database in the order that they were named. Our SQL query is written in uppercase, but it should be noted that SQL is a case-insensitive language. This means that it does not matter in what case we specify our SQL statements. We could have used the following statement and received the same results:

```
Select first_name, last_name
From authors;

FIRST_NAME            LAST_NAME
Tennessee             Williams
Steven                King
Danielle              Steeley
Margo                 Hennesay
Jordan                Michaels
```

It is important to remember, however, that the data within your database is case-sensitive. So, if we were to add a `WHERE` clause to our previous query in order to pull any authors with the last name of "King" from our database, it would be important to know in which case King was stored within the database. "King," "KING," "king," and "KinG" are all considered different from our `WHERE` clause.

```
// Incorrect case for King //
SELECT FIRST_NAME, LAST_NAME
FROM AUTHORS
WHERE LAST_NAME='king';

FIRST_NAME          LAST_NAME
No rows selected



// Correct case for King //
SELECT FIRST_NAME, LAST_NAME
FROM AUTHORS
WHERE LAST_NAME='King';

FIRST_NAME          LAST_NAME
Steven              King
```

There are other clauses that can also be added to our `SELECT` statement to further shape and restrict the data that is returned to us from our query. `DISTINCT` is used to restrict the query to return only distinct values and to drop all duplicates from the query.

```
SELECT DISTINCT <select_list> FROM <table_name>
```

`TOP N` is used to return the first *N* number of rows from the query results returned by the rest of the statement.

```
SELECT TOP <number or rows> <select_list> FROM <table_name>
```

`ORDER BY` is used to order the data from the query based upon the designated columns.

```
SELECT <select_list> FROM <table_name>
ORDER_BY <column_name1,column_name2,.......>
```

Using our previous example of the Authors table, we can combine several of these clauses to our `SELECT` statement to alter our result set.

```
SELECT TOP 2 FIRST_NAME, LAST_NAME
FROM AUTHORS
ORDER BY LAST_NAME;

FIRST_NAME          LAST_NAME
Margo               Hennesay
Steven              King
```

The last thing to note before we change our focus to the manipulation of data in SQL databases is that the query statements in this example use the semicolon as the terminating value. It should be noted that some implementations use this syntax and will not process a query statement without it (such as Oracle). However, others (such as SQL Server) make the syntax optional, so you can use it or not at your leisure. As always, it is best to check your specific version of RDBMS documentation to find out the proper syntax you should be using.

Manipulating Data in SQL Databases

Manipulating data within the database is centered on the use of the following three SQL statements:

- `INSERT`
- `UPDATE`
- `DELETE`

The `INSERT` statement handles the insertion of new data rows into the tables of your database. It can be called using either specific values or the `SELECT` statement to populate the new rows in your table. When you want to populate a single row in a table within your database, it is best to use the following version of the statement:

```
INSERT INTO <table_name> (<column_name1,column_name2,.....>)
VALUES (<value1,value2,........)
```

The most important things to remember about the `INSERT` statement are that the arguments in the column list and the value list must be equal in number, and that they must appear in the same order. So, using an example with our Authors table, we could use the following:

```
// Values are in incorrect order with respect to the Columns //
INSERT INTO Authors( First_Name, Last_Name)
VALUES ('King','Steven')

// Values are in the correct order with respect to the Columns //
INSERT INTO Authors( First_Name, Last_Name)
VALUES ('Steven','King')
```

This is a good example because it shows how the column names must align themselves with the values in both number and order. Another interesting fact is that both of these statements will execute because neither of them violates the structure of the table itself. The `Last_Name` and `First_Name` fields are obviously both character fields, and, as far as the database is concerned, they would both be valid values. This illustrates how careful the developer must be to make sure that the data being entered into the database is always entered in the correct fields to prevent complications down the road.

The other instance in which you would be entering data into a table would be one in which you wanted to insert multiple rows with a single SQL statement. In this case, you would use a `SELECT` statement in place of the `VALUES` statement.

```
INSERT INTO <table_name> (<column_name1, column_name2,.....>)
SELECT <select_list>
FROM <table_name>
```

In essence, you are inserting the results of your SELECT statement into the specified table within your database. In this instance, it is important to remember that the column list specified on the first line above must match the select list of the SELECT statement in both number and order. Failing to pay close attention to this will result in the possibility of incorrect values being loaded into your tables without an exception being raised, as detailed in our previous example.

Once your data is loaded into the tables of your database, it may become necessary to change some of the values. In order to accomplish this, you use the UPDATE statement.

```
UPDATE <table_name>
SET column_name1 = value1, column_name2 = value2, .......
WHERE
<search condition>
```

The UPDATE statement is called detailing which column values are to be changed, what they are to be changed to, and a search condition to limit the number of rows affected. Please remember that if you do not specify a search condition when issuing this command, then *all* of the rows within your table will be modified to the new value(s), possibly an unintended result. The following provides an example of the possible implementation of an UPDATE query on our theoretical Authors table:

```
// A query to check our original values //
SELECT FIRST_NAME, LAST_NAME FROM AUTHORS
WHERE LAST_NAME = 'King';
FIRST_NAME          LAST_NAME
Steven              King
// Now we update the First_Name of the row to Mike //
UPDATE Authors
SET First_Name = 'Mike'
WHERE
Last_Name = 'King';


// Now we query the table again to confirm the row has been changed //
SELECT FIRST_NAME, LAST_NAME FROM AUTHORS
WHERE LAST_NAME = 'King';
FIRST_NAME          LAST_NAME
Mike                King
```

At some point in time, it may become necessary to completely remove rows of old data from the tables within the database. To accomplish this, you would use the DELETE statement.

```
DELETE FROM <table_name>
WHERE <search condition>
```

The syntax of the DELETE statement is simple, but like the UPDATE statement, if you do not include the search condition, then *all* rows are affected. Following our previous example, the DELETE statement can be implemented like this:

```
// Verify that the row we are going to delete exists //
SELECT FIRST_NAME, LAST_NAME FROM AUTHORS
WHERE LAST_NAME = 'King';
FIRST_NAME          LAST_NAME
Mike                King
// Now we say goodbye to Mr. King //
DELETE
FROM Authors
WHERE
Last_Name = 'King';


// Let's verify that the row is gone //
SELECT FIRST_NAME, LAST_NAME FROM AUTHORS
WHERE LAST_NAME = 'King';
FIRST_NAME          LAST_NAME
No rows selected
```

Summary

You should now have a general understanding of the basics of SQL history and structure. SQL has been around since the late 1970s and has been set forth as the standard for the RDBMS industry. The standard is maintained by ANSI, but they no longer test for compliance with the standard. Considering that, you should always rely on your RDBMS documentation to determine if the specific implementation provides the level of compliance you desire.

In addition, you should also have a reasonable understanding of the major RDBMS implementations that will be used throughout this book. The differences between these systems will become more apparent as you traverse the chapters of this book. These differences are an important point to remember before taking the leap of deciding on a specific RDBMS implementation. These slight differences can lead to major problems if existing code must be ported to another vendor's implementation.

We have also detailed the basic functions (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) used to maintain the data within the tables of your database. We will use this basic knowledge as a building block in later chapters in order to show the power and functionality of using functions within your database's schema. In the next chapter, we will discuss the basic concepts of built-in and user-defined SQL functions, which comprise the majority of this book.