1

Relational Databases and SQL

As mentioned in the Introduction, SQL is the most widely used software tool for communicating with data residing in relational databases. In this endeavor, SQL utilizes elements of both language and logic. As a language, SQL employs a unique syntax with many English words, such as WHERE, FROM, and HAVING. As an expression of logic, it specifies the details of how data in a relational database is to be retrieved or updated.

With this duality in mind, we attempt in this book to emphasize both language and logic components as we present the topics that make up SQL. In all languages, whether they be computer or spoken, there are actual words to be learned and remembered. As such, we will present the various keywords present in SQL one at a time in a logical sequence. As we progress through each chapter, you'll build on your prior vocabulary to learn new keywords and exciting possibilities for interactions with a database.

In addition to the words themselves, there is also logic to be considered. The words employed by SQL have a distinct logical meaning and intent. The logic of SQL is just as important as the language. As in all computer languages, there is frequently more than one way to specify any desired objective. The nuances of what is possible encompass both the language and logic involved.

Let's start with the language. Once you become familiar with the syntax of SQL, you might find yourself thinking of SQL commands as analogous to English sentences and having a certain expressive meaning.

For example, compare this sentence:

```
I would like a hamburger and fries
from your value menu,
and make it to go.
```

with this SQL statement:

```
Select city, state
from Customers
order by state
```

We'll get into the details later, but this SQL statement means that we want the city and state fields from a table named Customers in a database, and we want the results sorted by state.

In both cases, we're specifying which items we want (hamburger/fries or city/state), where we want it from (value menu or Customers table), and some extra instructions (make it to go, or sort the results by state).

But before we get started, let's address one minor point: how to pronounce the word SQL. It turns out that there are two choices. One option is to simply to say it as individual letters, like "S-Q-L." Another possibility, preferred by the author, is to pronounce it as the word "sequel." This is one less syllable, and a little easier to say. However, there's no real agreement on the question. It's basically a matter of personal preference.

As for what the letters S-Q-L mean, most agree that they stand for "Structured Query Language." However, even here, there is not total agreement. Some would argue that SQL stands for nothing at all, since the language is derived from an old language from IBM called sequel, which did not, in fact, stand for structured query language.

What Is SQL?

So what is SQL? In a nutshell, SQL is a standard computer language for maintaining and utilizing data in relational databases. Put simply, SQL is a language that lets users interact with relational databases. It has a long history of development by various organizations going back to the 1970s. In 1986, the American National Standards Institute (ANSI) published its first set of standards regarding the language, and it has gone through several revisions since that time.

Generally speaking, there are three major components of the SQL language. The first is called *DML*, or *Data Manipulation Language*. This module of the language allows you to retrieve, update, add, or delete data in a database. The second component is called *DDL*, or *Data Definition Language*. DDL enables you to create and modify the database itself. For example, DDL provides ALTER statements that let you modify the design of tables in a database. Finally, the third component, *DCL*, or *Data Control Language*, maintains proper security for the database.

Major software vendors, such as Microsoft and Oracle, have adapted the standard for their own purposes and have added numerous extensions and modifications to the language. But although each vendor has its own unique interpretation of SQL, there is still an underlying base language, which is much the same for all vendors. That base language is what we'll cover in this book.

As a computer language, SQL is different from other languages you may be familiar with, such as Visual Basic or C++. These languages tend to be *procedural* in nature, meaning that they allow you to specify specific procedures to accomplish a desired task. SQL is more of a *declarative* language. In SQL, the desired objective is often declared with a single statement. The simpler structure of SQL is possible because it is concerned only with relational databases rather than the entirety of computer systems.

One additional point of clarification about the SQL language is that it is sometimes confused with specific SQL databases. Many software companies sell database management systems (DBMS) software. In common usage, the databases in these types of software packages are often referred to as *SQL databases*, because the SQL language is the primary means of managing and accessing data in these databases. Some vendors even use the word *SQL* as part of the database name. For example, Microsoft calls its latest database *SQL Server 2016*. But in fact, SQL is more properly a language than a database. Our focus in this book is on the language of SQL rather than on any particular database.

Microsoft SQL Server, MySQL, and Oracle

Although our aim is to cover the core language of SQL as it applies to all implementations, we must ultimately also provide specific examples of SQL syntax. And because syntax does vary somewhat among vendors, we've decided to focus on the SQL syntax utilized by these three popular databases:

- Microsoft SQL Server
- MySQL
- Oracle

In most cases, these databases have the same syntax. However, there are occasional differences. If there is any variance between these databases, the syntax for Microsoft SQL Server will be presented in the main text of this book. Any differences for MySQL or Oracle will be indicated in a sidebar titled "Database Differences," as shown here:

> **Database Differences**
>
> A sidebar such as this will appear whenever there are syntax differences for MySQL or Oracle. The syntax for Microsoft SQL Server will appear in the main text.

Microsoft SQL Server is available in several versions and editions. The most recent version is called *Microsoft SQL Server 2016*. Available editions run from a basic Express edition to a fully featured Enterprise edition. The Express edition is free but still has an abundance of features that allow users to get started with full-fledged database development. The Enterprise edition includes many sophisticated database management features, plus powerful business intelligence components.

Although owned by Oracle, MySQL is an open-source database, which means that no single organization controls its development. MySQL is available on numerous platforms other than Windows, such as Mac OS X and Linux. MySQL offers its Community Edition as a free download. The most recent version is MySQL 5.7.

The Oracle database is available in several editions. The most recent version is called *Oracle Database 12c*. The free version of the database is called the Express edition.

When starting out, it is sometimes useful to download the database of your choice, so you have something to experiment with. However, this book does not require you to do that. The material in this book has been written in such a way as to allow you to learn SQL simply by reading through the text. We'll provide enough data in the text so that you can understand the results of various SQL statements without having to download software and type in statements yourself.

Nevertheless, if you would like to download the free versions of any of these databases, we've included three appendixes with useful instructions and tips on how to do that. Appendix A has complete information on how to get started with Microsoft SQL Server. The instructions include details on how to install the software and execute SQL commands. Similarly, Appendixes B and C cover MySQL and Oracle.

As mentioned in the Introduction, our companion website provides supplemental material that lists all the SQL statements shown in this book in all three databases. However, it's likely that you'll find it unnecessary to download or view the additional material on the companion website. The examples shown throughout this book are self-explanatory and don't require you to do anything else in order to understand the material. However, if you are so inclined, feel free to take advantage of these extra features.

It should also be mentioned that, in addition to SQL Server, MySQL, and Oracle, other popular relational databases are worthy of consideration. For example:

- DB2, from IBM
- Informix, from IBM
- SQL Anywhere, from Sybase
- PostgreSQL, an open-source database
- Microsoft Access, from Microsoft

Of these databases, Microsoft Access is somewhat unique in that it has a graphical element. In essence, Access is a graphical interface for relational databases. In other words, Access allows you to create a query against a relational database entirely through graphical means. A useful aspect of Access for beginners is that you can easily create a query in a visual way and then switch to a SQL view to see the SQL statement you just created. Another distinction of Access is that it is primarily a desktop database. As such, you can use it to create a database that resides entirely in a single file on your PC, but it also allows you to connect to databases created with other tools, such as Microsoft SQL Server.

Relational Databases

With these preliminaries out of the way, let's now look at the basics of relational databases to see how they work. A relational database is a collection of data, stored in any number of tables. In common usage, the term *relational* can be taken to indicate that the tables are usually related to each other in some manner. However, in more precise terms, *relational* refers to

mathematical relation theory, and has to do with logical properties that govern the manner in which tables are related.

As an example, let's take the simple case of a database consisting of only two tables: Customers and Orders. The Customers table contains one record for each customer who ever placed an order. The Orders table has one record for each order. Each table can contain any number of fields, which are used to store the various attributes associated with each record. For example, a Customers table might contain fields such as FirstName and LastName.

At this point, it's useful to visualize some tables and the data they contain. The common custom is to display a table as a grid of rows and columns. Each row represents a record in the table. Each column represents a field in the table. The top header row normally contains the field names. The remaining rows show the actual data.

In SQL terminology, records and fields are referred to as *rows* and *columns*, corresponding to the visual representation. So henceforth, we'll use the terms *rows* and *columns* rather than *records* and *fields* to describe the design of tables in relational databases.

Let's look at an example of the simplest possible relational database. This database includes only two tables: Customers and Orders. This is what the Customers table might look like:

CustomerID	FirstName	LastName
1	Bob	Davis
2	Natalie	Lopez
3	Connie	King

The Orders table might appear as:

OrderID	CustomerID	OrderAmount
1	1	50.00
2	1	60.00
3	2	33.50
4	3	20.00

In this example, the Customers table contains three columns: CustomerID, FirstName, and LastName. There are currently three rows in the table, representing Bob Davis, Natalie Lopez, and Connie King. Each row represents a different customer, and each column represents a different piece of information about the customer. Similarly, the Orders table has three columns and four rows. This indicates that there are four orders in the database and three attributes for those orders.

Of course, this example is highly simplistic and only hints at the type of data that could be stored in a real database. For example, a Customers table would normally contain many additional columns describing other attributes of a customer, such as city, state, zip, and phone number. Similarly, an Orders table would ordinarily have columns describing additional attributes of the order, such as order date, sales tax, and the salesperson who took the order.

Primary and Foreign Keys

Note the first column in each table: CustomerID in the Customers table, and OrderID in the Orders table. These columns are commonly referred to as *primary keys*. Primary keys are useful and necessary for two reasons. First, they enable us to uniquely identify a single row in a table. For example, if we wanted to retrieve the row for Bob Davis, we could simply use the CustomerID column to obtain the data. Primary keys also ensure uniqueness. Designating the CustomerID column as a primary key guarantees that this column will have a unique value for every row in the table. Even if we happened to have two different men both named Bob Davis in our database, those rows would have different values in the CustomerID column.

In this example, the values in the primary key columns don't have any particular meaning. In the Customers table, the CustomerID column contains the values 1, 2, and 3 for the three rows in the table. Database tables are often designed in such a way as to generate sequential numbers automatically for the primary key column as new rows are added to the table. This design feature is usually referred to as *auto-increment*.

A second reason for primary keys is that they allow us to easily relate one table to another. In this example, the CustomerID column in the Orders table points to a corresponding row in the Customers table. Looking at the fourth row of the Orders table, notice that the CustomerID column has a value of 3. This means that this order is for the customer with a CustomerID of 3, who happens to be Connie King. The use of common columns among tables is an essential design element in relational databases.

In addition to merely pointing to the Customers table, the CustomerID column in the Orders table can be designated as something called a *foreign key*. We'll cover foreign keys in detail in Chapter 18, "Maintaining Tables," but for now, just be aware that foreign keys can be defined in order to ensure that the column has a valid value. As an example, you would not want the CustomerID column in the Orders table to have a particular value unless that CustomerID actually exists in the Customer table. The designation of a column as a foreign key can accomplish that restriction.

Datatypes

Primary and foreign keys add structure to a database table. They ensure that all tables in a database are accessible and properly related to each other. Another important attribute of every column in a table is its datatype.

Datatypes are simply a way of defining the type of data that the column can contain. A datatype must be specified for each column in every table. Unfortunately, there is a great deal of variation between relational databases as to which datatypes are allowed and what they mean. For example, Microsoft SQL Server, MySQL, and Oracle each have over 30 different allowable datatypes.

It would be impossible to cover the details and nuances of every available datatype, even for just these three databases. What we'll do, however, is summarize the situation by discussing the main categories of datatypes common to most databases. Once you understand the important datatypes in these categories, you will have little trouble with other datatypes you may encounter. Generally, there are three important kinds of datatypes: numeric, character, and date/time.

Numeric datatypes come in a variety of flavors, including bits, integers, decimals, and real numbers. Bits are numeric datatypes that allow for only two values: 0 and 1. Bit datatypes are often used to define an attribute as having a simple true or false type of value. Integers are numbers without decimal places. Decimal datatypes can contain decimal places. Unlike bits, integers, and decimals, real numbers are those numbers whose exact value is only approximately defined internally. The one distinguishing characteristic of all numeric datatypes is that they can be included in arithmetic calculations. Here are a few representative examples of numeric datatypes from Microsoft SQL Server, MySQL, and Oracle.

General Description	Microsoft SQL Server Datatype	MySQL Datatype	Oracle Datatype	Example
bit	bit	bit	(none)	1
integer	int	int	number	43
decimal	decimal	decimal	number	58.63
real	float	float	number	80.62345

Character datatypes are sometimes referred to as *string* or *character string* datatypes. Unlike numeric datatypes, character datatypes aren't restricted to numbers. They can include any alphabetic or numeric digit, and can even contain special characters, such as asterisks. When providing a value for character datatypes in SQL statements, the value must always be surrounded by single quotes. In contrast, numeric datatypes never use quotes. Here are a few representative examples of character datatypes:

General Description	Microsoft SQL Server Datatype	MySQL Datatype	Oracle Datatype	Example
variable length	varchar	varchar	varchar2	'Walt Disney'
fixed length	char	char	char	'60601'

The second example (60601) is presumably a zip code. At first glance, this looks like it might be a numeric datatype because it's composed only of numbers. This is not an unusual situation. Even though they contain only numbers, zip codes are usually defined as character datatypes because there is never a need to perform arithmetic calculations with zip codes.

Date/time datatypes are used for the representation of dates and times. Like character datatypes, date/time datatypes must be enclosed in single quotes. These datatypes allow for

special calculations involving dates. For example, a special function can be used to calculate the number of days between any two date/time dates. Here are a few examples of date/time datatypes:

General Description	Microsoft SQL Server Datatype	MySQL Datatype	Oracle Datatype	Example
date	date	date	(none)	'2017-02-15'
date and time	datetime	datetime	date	'2017-02-15 08:48:30'

NULL Values

Another important attribute of individual columns in a table is whether that column is allowed to contain null values. A null value means that there is no data for that particular data element. It literally contains no data. However, null values are not the same as spaces or blanks. Logically, null values and empty spaces are treated differently. The nuances of retrieving data that contains null values will be addressed in detail in Chapter 7, "Boolean Logic."

Many databases will display the word NULL in all capital letters when displaying data with null values. This is done so the user can tell that the data contains a null value and not simply spaces. We will follow that convention and display the word NULL throughout this book to emphasize that it represents that unique type of value.

Primary keys in a database can never contain NULL values. That is because primary keys, by definition, must contain unique values.

The Significance of SQL

Before leaving the general subject of relational databases, let's look at a brief historical overview in order to provide an appreciation of the usefulness of relational databases and the significance of SQL.

Back in the early days of computing in the 1960s, data was typically stored either on magnetic tape or in files on disk drives. Computer programs, written in languages such as FORTRAN and COBOL, typically read through input files and processed one record at a time, eventually moving data to output files. Processing was necessarily complex because procedures needed to be broken down into many individual steps involving temporary tables, sorting, and multiple passes through data until the desired output could be produced.

In the 1970s, advances were made as hierarchical and network databases were invented and utilized. These newer databases, through an elaborate system of internal pointers, made it easier to read through data. For example, a program could read a record for a customer, automatically be pointed to all orders for that customer, and then to all details for each order. But basically that data still needed to be processed one record at a time.

The main problem with data storage prior to relational databases was not how the data was stored, but how it was accessed. The real breakthrough with relational databases came when the language of SQL was developed, because it allowed for an entirely new method of accessing data.

Unlike earlier data retrieval methods, SQL permitted the user to access a large set of data at once. With a single statement, a SQL command could retrieve or update thousands of records from multiple tables. This eliminated a great deal of complexity. Computer programs no longer needed to read one record at a time in a special sequence, while deciding what to do with each record. What used to require hundreds of lines of programming code could now be accomplished with just a few lines of logic.

Looking Ahead

This first chapter provided some background information about relational databases, allowing us to move on to the main topic of retrieving data from databases. We discussed a number of important characteristics of relational databases, such as primary keys, foreign keys, and datatypes. We also talked about the possible existence of NULL values in data. We'll add to our discussion of NULL values in Chapter 7, "Boolean Logic," and return to the general topics of database maintenance in Chapter 18, "Maintaining Tables," and database design in Chapter 19, "Principles of Database Design."

Why is the important topic of database design postponed until much later in this book? In short, this approach is taken so you can plunge into using SQL without having to worry about the details of design at the beginning. In truth, database design is as much an art as it is a science. The principles of database design will hopefully be much more meaningful after you've become more aware of the details and nuances of retrieving some data via SQL. Therefore, we'll temporarily ignore the question of how to design a database and commence with data retrieval in our next chapter.

2

Basic Data Retrieval

Keywords Introduced

SELECT · FROM

In this chapter, we'll begin our exploration of the most important topic in SQL—namely, how to retrieve data from a database. Regardless the size of your organization, the most common request made of SQL developers is the request for a report. Of course, it's a nontrivial exercise to get data into a database, but once the data is there, the energies of business analysts turn to the wealth of data at their disposal and the desire to extract useful information from all that data.

The emphasis in this book on data retrieval corresponds to these real-world demands. Your knowledge of SQL will go a long way toward helping your organization unlock the secrets hidden in the data stored in your databases.

A Simple SELECT

The ability to retrieve data in SQL is accomplished through something called the SELECT statement. Without any preliminary explanation, here is an example of the simplest possible SELECT statement:

```
SELECT * FROM Customers
```

In the SQL language, as in all computer languages, certain words are *keywords*. These words have special meanings and must be used in a particular way. In this statement, the words SELECT and FROM are keywords. The SELECT keyword indicates the start of a SELECT statement. The FROM keyword is used to designate the table from which data is to be retrieved. The name of the table follows the FROM. In this case, the table name is Customers. The asterisk (*) in this example is a special symbol that means "all columns."

As is custom, we'll print keywords in all capital letters. This is done to ensure that they are noticeable. To sum up, this statement means: Select all columns from the Customers table.

If the Customers table looks like this:

CustomerID	FirstName	LastName
1	Sara	Davis
2	Rumi	Shah
3	Paul	Johnson
4	Samuel	Martinez

then the SELECT will return the following data:

CustomerID	FirstName	LastName
1	Sara	Davis
2	Rumi	Shah
3	Paul	Johnson
4	Samuel	Martinez

In other words, it brings back everything in the table.

In the previous chapter, we mentioned that it's a common practice to specify a primary key for all tables. In this example, the CustomerID column is such a column. We also mentioned that primary keys are sometimes set up to automatically generate sequential numbers in a numeric sequence as rows are added to a table. That is the case in this example. In fact, most of the sample data we'll show throughout the book will include a similar column that is both a primary key and defined as auto-increment. By convention, this is generally the first column in a table.

Syntax Notes

Two points must be remembered when writing any SQL statement. First, the keywords in SQL are not case sensitive. The word SELECT is treated identically to "select" or "Select."

Second, a SQL statement can be written on any number of lines and with any number of spaces between words. For example, the SQL statement:

```
SELECT * FROM Customers
```

is identical to:

```
SELECT *
FROM Customers
```

It's usually a good idea to begin each important keyword on a separate line. When we get to more complex SQL statements, this will make it easier to quickly grasp the meaning of the statement.

Finally, as we present different SQL statements in this book, we'll often show both a specific example and a more general format. For instance, the general format of the previous statement would be shown as:

```
SELECT *
FROM table
```

Italics are used to indicate a general expression. The italicized word *table* means that you can substitute any table name in that spot. When you see italicized words in any SQL statement in this book, that is simply a way of indicating that any valid word or phrase can be substituted in that location.

Database Differences: MySQL and Oracle

Many SQL implementations require a semicolon at the end of every statement. This is true of MySQL and Oracle, but not of Microsoft SQL Server. However, semicolons can be specified in SQL Server if desired. For simplicity, we'll show SQL statements without semicolons in this book. If you're using MySQL or Oracle, you'll need to add a semicolon at the end of each statement. The previous statement would appear as:

```
SELECT *
FROM Customers;
```

Comments

When writing SQL statements, it's often desirable to insert comments within or around those statements. There are two standard methods of writing comments in SQL. The first method, referred to as the double dash, consists of two dashes placed anywhere on a line. All text that follows two dashes on that line is ignored and is interpreted as comments. Here's an example of this format:

```
SELECT
-- this is the first comment
FirstName,
LastName -- this is a second comment
FROM Customers
```

The second format, borrowed from the C programming language, consists of text placed between /* and */ characters. Comments between the /* and */ can be written on multiple lines. Here's an example:

```
SELECT
/* this is the first comment */
FirstName,
LastName /* this is a second comment
this is still part of the second comment
this is the end of the second comment */
FROM Customers
```

Database Differences: MySQL

MySQL supports comments in both the double dash and the C programming format (/* and */), with one minor difference. When using the double dash format, MySQL requires a space or special character such as a tab immediately after the second dash.

In addition, MySQL allows for a third method of inserting comments, similar to the double dash. In MySQL, you can place a number sign (#) anywhere on a line to indicate comments. All text after the # symbol on that line is taken as a comment. Here's an example of this format:

```
SELECT FirstName
# this is a comment
FROM Customers;
```

Specifying Columns

So far, we've done nothing more than simply display all the data in a table. But what if we wanted to select only certain columns? Working from the same table, we might want to display only the customer's last name, for example. The SELECT statement would then look like:

```
SELECT LastName
FROM Customers
```

and the resulting data would be:

LastName
Davis
Shah
Johnson
Martinez

If we wanted to select more than one, but not all, columns, the SELECT might look like this:

```
SELECT
FirstName,
LastName
FROM Customers
```

and the output would appear as:

FirstName	LastName
Sara	Davis
Rumi	Shah
Paul	Johnson
Samuel	Martinez

The general format of this statement is:

```
SELECT columnlist
FROM table
```

The important thing to remember is that if you need to specify more than one column in the *columnlist*, then those columns must be separated by a comma. Also notice that we placed each column (FirstName, LastName) in the *columnlist* on separate lines. This was done to improve readability.

Column Names with Embedded Spaces

What if a column contains a space in its name? Say, for example, that the LastName column was specified as Last Name (with a space between the two words). Clearly, the following would not work:

```
SELECT
Last Name
FROM Customers
```

This statement would be considered invalid because Last and Name are not column names, and even if they were proper column names, they would need to be separated by a comma. The solution is to place special characters around any column name containing spaces. The character to use differs, depending on which database you're using. For Microsoft SQL Server, the required characters are square brackets. The syntax is:

```
SELECT
[Last Name]
FROM Customers
```

One additional syntax note: Just as keywords are not case sensitive, it's also true that table and column names are not case sensitive. As such, the previous example is identical to:

```
select
[last name]
from customers
```

For clarity's sake, we'll print all keywords in caps, and we'll also capitalize table and column names in this book, but that is not truly necessary.

Database Differences: MySQL and Oracle

For MySQL, the character to use around column names containing spaces is an accent grave (`). The MySQL syntax for the above example is:

```
SELECT
`Last Name`
FROM Customers;
```

For Oracle, the character to use around column names containing spaces is the double quote. The Oracle syntax for the example is:

```
SELECT
"Last Name"
FROM Customers;
```

Additionally, unlike Microsoft SQL Server and MySQL, column names in Oracle surrounded by double quotes are case sensitive. This means that the previous statement is not equivalent to:

```
SELECT
"LAST NAME"
FROM Customers;
```

Preview of the Full SELECT

The bulk of this book has to do with the SELECT statement introduced in this chapter. In Chapters 3 through 15, we'll expand on this statement, introducing new features until the full potential and capabilities of the SELECT are realized and understood. At this point, we have only introduced this portion of the SELECT statement:

```
SELECT columnlist
FROM table
```

In the interest of removing any remaining suspense, let's look at a preview of the full SELECT statement and briefly comment on its various components. The full SELECT statement, with all its clauses is:

```
SELECT columnlist
FROM tablelist
WHERE condition
GROUP BY columnlist
HAVING condition
ORDER BY columnlist
```

We've already been introduced to the SELECT and FROM clauses. Let's expand a bit on those clauses and talk about the others. The SELECT clause initiates the statement and lists any columns that will be displayed. As will be seen in later chapters, the *columnlist* can include not only actual columns from the specified tables, but also calculated columns, usually derived from one or more columns in the tables. The columns in the *columnlist* can also include functions, which represent a special way of adding commonly used methods for transforming data.

The FROM clause specifies the data sources from which data will be drawn. In most cases, these data sources will be tables. In later chapters, we'll learn that these data sources can also be other SELECT statements, which represent a type of virtual *view* of data. In this chapter, our *tablelist* is a single table. One of the key features of SQL to be discussed in later chapters is the ability to combine multiple tables together in a single SELECT statement through something called the JOIN. Thus, we'll see many examples where the *tablelist* in the FROM clause is composed of multiple lines, indicating tables joined together.

The WHERE clause is used to indicate selection logic. This is where you can specify exactly which rows of data are to be retrieved. The WHERE clause can utilize basic arithmetic operators such as equals (=) and greater than (>), along with Boolean operators such as OR and AND.

The GROUP BY clause plays a key role in summarizing data. By organizing data into various groups, the analyst has the ability to not only group data, but to summarize the data in each group through the use of various statistics, such as a sum or count of the data.

When data has been grouped, selection criteria become somewhat more complex. One has to ask whether the selection criteria apply to individual rows or to the entire group. For example, when grouping customers by state, one may only want to see rows of individual customers where the aggregate purchases of all customers in the state exceed a certain amount. This is where the HAVING clause comes in. The HAVING clause is used to specify selection logic for an entire group of data.

Finally, the ORDER BY clause is used to sort the data in an ascending or descending sequence.

As will be made clear in later chapters, the various clauses in a SELECT statement, if they exist, must be specified in the same order shown in the previous general statement. For example, if there is a GROUP BY clause in a SELECT statement, it must appear after a WHERE clause and before a HAVING clause.

In addition to all of the aforementioned clauses, we will also talk about a number of additional ways to organize the SELECT statement, including subqueries and set logic. Subqueries are a way to insert an entire SELECT statement within another SELECT statement, and are often useful for certain types of selection logic. Set logic is a way to combine multiple queries side by side as a single query.

Looking Ahead

In this chapter, we began our exploration of how to use the SELECT statement to retrieve data. We learned about basic syntax and have seen how to select specific columns. In reality, however, this allows us to accomplish very little of a practical nature. Most significantly, we have not yet learned how to apply any type of selection criteria to our data retrieval efforts. For example, while we know how to select all customers, we don't yet know how to select only customers from the state of New York.

As it happens, we won't cover selection criteria until Chapter 6. What will we be doing until then? In the next few chapters, we'll build on what can be done with the *columnlist* component of the SELECT statement. In the following chapter, we'll move on to more variations on column selection, allowing us to create complex calculations in a single column. We'll also talk about ways to rename columns to make them more descriptive. Chapters 4 and 5 will then build on our ability to create even more complex and powerful *columnlists*, so when we finally get to the topic of selection criteria in Chapter 6, we'll have a full arsenal of techniques available at our disposal.