5

Sorting Data

Keywords Introduced

ORDER BY · ASC · DESC

The ability to present data in a sorted order is often essential to the task at hand. For example, if an analyst is shown a large list of customers in a random order, they'd probably find it difficult to locate one particular customer. However, if the same list is sorted alphabetically, the desired customer can quickly be located.

The idea of sorting data applies to many situations, even when the data isn't alphabetic in nature. For example, orders can be sorted by order date, allowing one to rapidly find an order taken at a particular date and time. Alternatively, orders can be sorted by the order amount, allowing orders to be viewed from the smallest to largest. No matter what particular form a sort takes, it can add a useful way of organizing the data being presented to an end user.

Sorting in Ascending Order

Up until now, data has not been returned in any particular order. When a SELECT is issued, you never know which row will come first. If the query is executed from within a software program, and no one ever sees the data at that point in time, then it really doesn't matter. But if the data is to be immediately displayed to a user, then the order of rows is often significant. A sort can be easily added to a SELECT statement by using an ORDER BY clause.

Here's the general format for a SELECT statement with an ORDER BY clause:

```
SELECT columnlist
FROM tablelist
ORDER BY columnlist
```

The ORDER BY clause is always placed after the FROM clause, which in turn always comes after the SELECT keyword. The italicized *columnlist* for the SELECT and ORDER BY keywords indicates that any number of columns can be listed. The columns in *columnlist* can be individual columns or more complex expressions. Also, the columns specified after the SELECT and ORDER BY keywords can be entirely different columns. The italicized *tablelist* indicates that

any number of tables can be listed in the FROM clause. The syntax for listing multiple tables will be introduced in Chapter 11, "Inner Joins," and Chapter 12, "Outer Joins."

For the following few examples on sorting, we'll work from data in this Salespeople table:

SalespersonID	FirstName	LastName
1	Gregory	Brown
2	Carla	Brown
3	Natalie	Lopez
4	Connie	King

To sort data in an alphabetic order, with A coming before Z, we simply need to add an ORDER BY clause to the SELECT statement. For example:

```
SELECT
FirstName,
LastName
FROM Salespeople
ORDER BY LastName
```

brings back this data:

FirstName	LastName
Gregory	Brown
Carla	Brown
Natalie	Lopez
Connie	King

Because there are two Browns, Carla and Gregory, there's no way to predict which one will be listed first. This is because we are sorting only on LastName, and there are multiple rows with that same last name.

Similarly, if we issue this SELECT:

```
SELECT
FirstName,
LastName
FROM Salespeople
ORDER BY FirstName
```

then this data is retrieved:

FirstName	LastName
Carla	Brown
Connie	King
Gregory	Brown
Natalie	Lopez

The order is now completely different, because the sort is by first name rather than last name.

SQL provides a special keyword named ASC, which stands for *ascending*. This keyword is completely optional and largely unnecessary, because all sorts are assumed to be in ascending order by default. The following SELECT, which uses the ASC keyword, returns the same data shown previously.

```
SELECT
FirstName,
LastName
FROM Salespeople
ORDER BY FirstName ASC
```

In essence, the keyword ASC can be used to emphasize the fact that the sort is in ascending, rather than descending, order.

Sorting in Descending Order

The DESC keyword sorts in an order opposite to ASC. Instead of ascending, the order in such a sort is descending. For example:

```
SELECT
FirstName,
LastName
FROM Salespeople
ORDER BY FirstName DESC
```

retrieves:

FirstName	LastName
Natalie	Lopez
Gregory	Brown
Connie	King
Carla	Brown

The first names are now in a Z-to-A order.

Sorting by Multiple Columns

We now return to the problem of what to do with the Browns. To sort by last name when there is more than one person with the same last name, we must add a secondary sort by first name, as follows:

```
SELECT
FirstName,
LastName
FROM Salespeople
ORDER BY LastName, FirstName
```

This brings back:

FirstName	LastName
Carla	Brown
Gregory	Brown
Connie	King
Natalie	Lopez

Because a second sort column is now specified, we can now be certain that Carla Brown will appear before Gregory Brown. Note that LastName must be listed before FirstName in the ORDER BY clause. The order of the columns is significant. The first column listed always has the primary sort value. Any additional columns listed become secondary, tertiary, and so on.

Sorting by a Calculated Field

We'll now apply our knowledge of calculated fields and aliases from Chapter 3 to illustrate some further possibilities for sorts. This statement:

```
SELECT
LastName + ', ' + FirstName AS 'Name'
FROM Salespeople
ORDER BY Name
```

returns this data:

Name
Brown, Carla
Brown, Gregory
King, Connie
Lopez, Natalie

As seen, we utilized concatenation to create a calculated field with an alias of Name. We are able to refer to that same column alias in the ORDER BY clause. This nicely illustrates another benefit of using column aliases. Also, note the design of the calculated field itself. We inserted a column and a space between the last name and first name columns to separate them, and to show the name in a commonly used format. Conveniently, this format also works well for sorting. The ability to display names in this format, with a comma separating the last and first name, is a handy trick to keep in mind. Users very often want to see names arranged in this manner.

It's also possible to put a calculated field in the ORDER BY clause without also using it as a column alias. Similar to the above, we could also specify:

```
SELECT
FirstName,
LastName
FROM Salespeople
ORDER BY LastName + FirstName
```

This would display:

FirstName	LastName
Carla	Brown
Gregory	Brown
Connie	King
Natalie	Lopez

The data is sorted the same as in the prior example. The only difference is that we're now specifying a calculated field in the ORDER BY clause without making use of column aliases. This gives the same result as if LastName and FirstName were specified as the primary and secondary sort columns.

Sort Sequences

In the previous examples, all of the data is character data, consisting of letters from A to Z. There are no numbers or special characters. Additionally, there has been no consideration of upper- and lowercase letters.

Every database lets users specify or customize collation settings that provide details on how data is sorted. The settings vary among databases, but three facts are generally true. First, when data is sorted in an ascending order, any data with NULL values appear first. As previously discussed, NULL values are those where there is an absence of data. After any NULLs, numbers will appear before characters. For data sorted in descending order, character data will display first, then numbers, and then NULLs.

Second, for character data, there is usually no differentiation between upper- and lowercase. An e is the treated the same as an E. Third, for character data, the individual characters that make up the value are evaluated from left to right. If we're talking about letters, then AB will come before AC. Let's look at an example, taken from this table, which we'll refer to as TableForSort:

TableID	CharacterData	NumericData
1	23	23
2	5	5
3	Dog	NULL
4	NULL	–6

In this table, the CharacterData column is defined as a character column, for example as VARCHAR (a variable length datatype). Similarly, the NumericData column is defined as a numeric column, such as INT (an integer datatype). Values with no data are displayed as NULL. When this SELECT is issued against the TableForSort table:

```
SELECT
NumericData
FROM TableForSort
ORDER BY NumericData
```

it will display:

NumericData
NULL
-6
5
23

Notice that NULLs come first, then the numbers in numeric sequence. If we want the NULL values to assume a default value of 0, we can use the ISNULL function discussed in the previous chapter and issue this SELECT statement:

```
SELECT
ISNULL(NumericData,0) AS 'NumericData'
FROM TableForSort
ORDER BY ISNULL(NumericData,0)
```

The result is now:

NumericData
-6
0
5
23

The ISNULL function converted the NULL value to a 0, which results in a different sort order.

The decision as to whether to display NULL values as NULL or as 0 depends on the specific circumstance. If the user thinks of NULL values as meaning 0, then they should be displayed as 0. However, if the user sees NULL values as an absence of data, then displaying the word NULL is appropriate.

Turning to a different ORDER BY clause against the same table, if we issue this SELECT:

```
SELECT
CharacterData
FROM TableForSort
ORDER BY CharacterData
```

it will display:

CharacterData
NULL
23
5
Dog

As expected, NULLs come first, then values with numeric digits, and then values with alphabetic characters. Notice that 23 comes before 5. This is because the 23 and 5 values are being evaluated as characters, not as numbers. Because character data is evaluated from left to right and 2 is lower than 5, 23 is displayed first.

Looking Ahead

In this chapter, we talked about the basic possibilities for sorting data in a specific order. We illustrated how to sort by more than one column. We also discussed the use of calculated fields in sorts. Finally, we covered some of the quirks of sorting, particularly when it comes to data with NULL values and with numbers in character columns.

At the beginning of the chapter, we mentioned some of the general uses for sorts. Primary among these is the ability to simply place data in an easily understood order, thus allowing users to quickly locate a desired piece of information. People generally like to see data in some kind of order, and sorts accomplish that goal. Another interesting use of sorts will be covered in Chapter 6, "Selection Criteria." In that chapter, we'll introduce the keyword TOP and another way to use sorts in conjunction with that keyword. This technique, commonly known as a Top N sort, allows us to do such things as display customers with the five highest orders for a given time period.

In our next chapter, we'll move beyond our analysis of what can be done with *columnlists* and discuss data selection. The ability to specify selection criteria in SELECT statements is critical to most normal queries. In the real world, it would be very unusual to issue a SELECT statement without some sort of selection criteria. The topics discussed in the next chapter address this important topic.

6 Selection Criteria

Keywords Introduced

WHERE · TOP · LIKE

Up until this point, the SELECT statements we've seen have always brought back every row in the table. This would rarely be the case in real-world situations. One is normally interested only in retrieving data that meets certain criteria. For example, if you're selecting orders, you probably only want to see orders that meet certain conditions. When looking at products, you ordinarily only want to view certain types of products. Rarely does someone want to simply see everything. Your interest in data is typically directed toward a small subset of that data in order to analyze or view one particular aspect.

Applying Selection Criteria

Selection criteria in SQL begins with the WHERE clause. The WHERE keyword accomplishes the task of selecting a subset of rows. This is the general format of the SELECT statement, including the WHERE clause and other clauses previously discussed:

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

As can be seen, the WHERE clause must always come between the FROM and ORDER BY clauses. In fact, if any clause is used, it must appear in the order shown above.

Let's look at an example, taken from data in this Sales table:

SalesID	FirstName	LastName	QuantityPurchased	PricePerItem
1	Andrew	Li	4	2.50
2	Carol	White	10	1.25
3	James	Carpenter	5	4.00

We'll start with a statement with a simple WHERE clause:

```
SELECT
FirstName,
LastName,
QuantityPurchased
FROM Sales
WHERE LastName = 'Carpenter'
```

The output is:

FirstName	LastName	QuantityPurchased
James	Carpenter	5

Because the WHERE clause stipulates to select only rows with a LastName equal to 'Carpenter', only one of the three rows in the table is returned. Notice that the desired value of the LastName column was enclosed in quotes, due to the fact that LastName is a character column. For numeric fields, no quotes are necessary. For example, the following SELECT is equally valid and returns the same data:

```
SELECT
FirstName,
LastName,
QuantityPurchased
FROM Sales
WHERE QuantityPurchased = 5
```

WHERE Clause Operators

In the previous statements, an equals sign (=) is used as the operator in the WHERE clause. The equals sign indicates a test for equality. The general format shown above requires that a *condition* follows the WHERE clause. This condition consists of an operator with expressions on either side.

The following is a list of the basic operators that can be used in the WHERE clause:

WHERE Operator	Meaning
=	equals
<>	does not equal
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

More advanced operators will be covered in the next chapter.

The meaning of the equals (=) and does not equal (<>) operators should be obvious. Here's an example of a WHERE clause with an "is greater than" operator, taken from the same Sales table:

```
SELECT
FirstName,
LastName,
QuantityPurchased
FROM Sales
WHERE QuantityPurchased > 6
```

The result is:

FirstName	LastName	QuantityPurchased
Carol	White	10

In this example, only one row meets the condition that the QuantityPurchased column be greater than 6. Although not as commonly used, it is also possible to use the "is greater than" operator with a text column. This example:

```
SELECT
FirstName,
LastName
FROM Sales
WHERE LastName > 'K'
```

returns:

FirstName	LastName
Andrew	Li
Carol	White

Because the test is for last names greater than K, it brings back only Li and White, but not Carpenter. When applied to text fields, the greater than and less than operators indicate selection by the alphabetic order of the values. In this case, Li and White are returned, since L and W come after K in the alphabet.

Limiting Rows

We may sometimes want to select a small subset of the rows in a table, but don't care which rows are returned. Let's say we have a table with 50,000 rows and want to see just a few rows of data to see what it looks like. It wouldn't make sense to use the WHERE clause for this purpose, since we don't care which particular rows are returned.

For this situation, the solution is to use a special keyword to specify a limit as to how many rows are returned. This is another instance where syntax differs among databases. In Microsoft SQL Server, the keyword that accomplishes this limit is TOP. The general format is:

```
SELECT
TOP number columnlist
FROM tablelist
```

Database Differences: MySQL and Oracle

MySQL uses the keyword LIMIT rather than TOP. The general format is:

```
SELECT columnlist
FROM tablelist
LIMIT number
```

Oracle uses the keyword ROWNUM rather than TOP. The ROWNUM keyword must be specified in a WHERE clause, as follows:

```
SELECT columnlist
FROM tablelist
WHERE ROWNUM <= number
```

Let's say that we want to see the first 10 rows from a table. The SELECT to accomplish this looks like:

```
SELECT
TOP 10 *
FROM table
```

This statement returns all columns in the first 10 rows from the table. Like any SELECT statement without an ORDER BY clause, there's no way to predict which 10 rows will be returned. It depends on how the data is physically stored in the table.

Similarly, we can list specific columns to return:

```
SELECT
TOP 10
column1,
column2
FROM table
```

In essence, the TOP keyword accomplishes something similar to the WHERE clause, as it returns a small subset of rows in the specified table. However, keep in mind that rows returned using the TOP keyword are not a true random sample, in a statistical sense. They're only the first rows that qualify, based on how the data is physically stored in the database.

Limiting Rows with a Sort

Another use of the TOP keyword is to use it in combination with the ORDER BY clause to obtain a designated number of rows with the highest values, based on specified criteria. This type of data selection is commonly referred to as a *Top N* selection. Here's an example, taken from this Books table:

BookID	Title	Author	CurrentMonthSales
1	Pride and Prejudice	Austen	15
2	Animal Farm	Orwell	7
3	Merchant of Venice	Shakespeare	5
4	Romeo and Juliet	Shakespeare	8
5	Oliver Twist	Dickens	3
6	Candide	Voltaire	9
7	The Scarlet Letter	Hawthorne	12
8	Hamlet	Shakespeare	2

Let's say we want to see the three books that sold the most in the current month. The SELECT that accomplishes this is:

```
SELECT
TOP 3
Title AS 'Book Title',
CurrentMonthSales AS 'Quantity Sold'
FROM Books
ORDER BY CurrentMonthSales DESC
```

The output is:

Book Title	Quantity Sold
Pride and Prejudice	15
The Scarlet Letter	12
Candide	9

Let's examine this statement in some detail. The TOP 3 in the second line indicates that only three rows of data are to be returned. The main question to ask is how it determines which three rows to display. The answer is found in the ORDER BY clause. If there were no ORDER BY clause, then the SELECT would simply bring back any three rows of data. However, this is not what we want. We're looking for the three rows with the highest sales. To accomplish this, we need to sort the rows by the CurrentMonthSales column in descending order. Why descending? When data is sorted in descending order, the highest numbers appear first. If we had sorted in an ascending order, we would get the books with the least number of sales, not the most.

Let's now add one more twist to this scenario. Let's say we only want to see which book by Shakespeare has sold the most. In order to accomplish this, we need to add a WHERE clause, as follows:

```
SELECT
TOP 1
Title AS 'Book Title',
CurrentMonthSales AS 'Quantity Sold'
FROM Books
WHERE Author = 'Shakespeare'
ORDER BY CurrentMonthSales DESC
```

This brings back this data:

Book Title	Quantity Sold
Romeo and Juliet	8

The WHERE clause adds the qualification to look only at books by Shakespeare. We also revised the TOP keyword to specify TOP 1, indicating that we want to see only one row of data.

Pattern Matching

We now want to turn to a situation in which the data to be retrieved is not precisely defined. We often want to view data based on inexact matches with words or phrases. For example, you might be interested in finding companies whose name contains the word "bank." The selection of data via inexact matches within phrases is often referred to as *pattern matching*. In SQL, the LIKE operator is used in the WHERE clause to enable us to find matches against part of a column value. The LIKE operator requires the use of special wildcard characters to specify exactly how the match should work. Let's start with an example from the following Movies table.

MovieID	MovieTitle	Rating
1	Love Actually	R
2	North by Northwest	Not Rated
3	Love and Death	PG
4	The Truman Show	PG
5	Everyone Says I Love You	R
6	Down with Love	PG-13
7	Finding Nemo	G

Our first example with a LIKE operator is:

```
SELECT
MovieTitle AS 'Movie'
FROM Movies
WHERE MovieTitle LIKE '%LOVE%'
```

In this example, the percent (%) sign is used as a wildcard. The percent (%) wildcard means *any characters*. *Any characters* includes the possibility of there being no characters. The percent (%) before LOVE means that we will accept a phrase with any (or no) characters before LOVE. Similarly, the percent (%) after LOVE means that we'll accept a phrase with any (or no) characters after LOVE. In other words, we're looking for any movie title that contains the word LOVE. Here is the data returned from the SELECT:

Movie
Love Actually
Love and Death
Everyone Says I Love You
Down with Love

Notice that LOVE appears as the first word, the last word, and sometimes in the middle of the movie title.

Database Differences: Oracle

Unlike Microsoft SQL Server and MySQL, Oracle is case sensitive when determining matches for literal values. In Oracle, LOVE is not the same as Love. An equivalent statement in Oracle is:

```
SELECT
MovieTitle AS Movie
FROM Movies
WHERE MovieTitle LIKE '%LOVE%';
```

This would return no data, because no movie title contains the word LOVE in all uppercase. One solution in Oracle is to use the UPPER function to convert your data to uppercase, as follows:

```
SELECT
MovieTitle AS Movie
FROM Movies
WHERE UPPER(MovieTitle) LIKE '%LOVE%';
```

Let's now attempt to find only movies that begin with LOVE. If we issue:

```
SELECT
MovieTitle AS 'Movie'
FROM Movies
WHERE MovieTitle LIKE 'LOVE%'
```

we will retrieve only this data:

Movie
Love Actually
Love and Death

Because we're now specifying the percent (%) wildcard only after the word LOVE, we get back only movies that begin with LOVE. Similarly, if we issue:

```
SELECT
MovieTitle AS 'Movie'
FROM Movies
WHERE MovieTitle LIKE '%LOVE'
```

we get only this data:

Movie
Down with Love

This is because we have now specified that the phrase must *end* with the word LOVE.

One might ask how to arrange the wildcards to see only movies that contain the word LOVE in the middle of the title, without seeing movies where LOVE is at the beginning or end. The solution is to specify:

```
SELECT
MovieTitle AS 'Movie'
FROM Movies
WHERE MovieTitle LIKE '% LOVE %'
```

Notice that a space has been inserted between the word LOVE and the percent (%) wildcards on either side. This ensures that there is at least one space on both sides of the word. The data brought back from this statement is:

Movie
Everyone Says I Love You

Wildcards

The percent (%) sign is the most common wildcard used with the LIKE operator, but there are a few other possibilities. These include the underscore character (_), a *characterlist* enclosed in square brackets, and a caret symbol (^) plus a *characterlist* enclosed in square brackets. The following table lists these wildcards and their meanings:

Wildcard	Meaning
%	any characters (can be zero characters)
_	exactly one character (can be any character)
[*characterlist*]	exactly one character in the *characterlist*
[^*characterlist*]	exactly one character *not* in the *characterlist*

We'll use the following Actors table to illustrate statements for these wildcards.

ActorID	FirstName	LastName
1	Cary	Grant
2	Mary	Steenburgen
3	Jon	Voight
4	Dustin	Hoffman
5	John	Wayne
6	Gary	Cooper

Here's an illustration of how the underscore (_) wildcard character can be used:

```
SELECT
FirstName,
LastName
FROM Actors
WHERE FirstName LIKE '_ARY'
```

The output of this SELECT is:

FirstName	LastName
Cary	Grant
Mary	Steenburgen
Gary	Cooper

This statement retrieves these three actors because all have a first name consisting of exactly one character followed by the phrase ARY.

Likewise, if we issue this statement:

```
SELECT
FirstName,
LastName
FROM Actors
WHERE FirstName LIKE 'J_N'
```

it produces:

FirstName	LastName
Jon	Voight

The actor John Wayne is not selected since John doesn't fit the J_N pattern. An underscore stands for only one character.

The final wildcards we'll discuss, [*characterlist*] and [^*characterlist*], enable you to specify multiple wildcard values in a single position.

Database Differences: MySQL and Oracle

The [*characterlist*] and [^*characterlist*] wildcards are not available in MySQL or Oracle.

The following illustrates the [*characterlist*] wildcard:

```
SELECT
FirstName,
LastName
FROM Actors
WHERE FirstName LIKE '[CM]ARY'
```

This retrieves any rows where FirstName begins with a C or M and ends with ARY. The result is:

FirstName	LastName
Cary	Grant
Mary	Steenburgen

The following illustrates the [^*characterlist*] wildcard:

```
SELECT
FirstName,
LastName
FROM Actors
WHERE FirstName LIKE '[^CM]ARY'
```

This selects any rows where FirstName does not begin with a C or M and ends with ARY. The result is:

FirstName	LastName
Gary	Cooper

Looking Ahead

This chapter introduced the topic of how to apply selection criteria to queries. A number of basic operators, such as equals and greater than, were introduced. The ability to specify these types of basic selection criteria goes a long way toward making the SELECT statement truly useful. We also covered the related topic of limiting the number of rows returned in a query. The ability to limit rows in combination with an ORDER BY clause allows for a useful Top N type of data selection.

We concluded the chapter with a study of matching words or phrases via a specified pattern. Matching by patterns is a significant and widely used function of SQL. Any time you enter a word in a search box and attempt to retrieve all entities containing that word, you are utilizing pattern matching.

In our next chapter, "Boolean Logic," we'll greatly enhance our selection criteria capabilities by introducing a number of new keywords that add sophisticated logic to the WHERE clause. At present, we can do such things as select all customers from the state of New York. In the real world, however, much more is typically required. Boolean logic will allow us to formulate a query that will select customers who are in New York or California but not in Los Angeles or Albuquerque.