2.3 WHERE and ORDER BY (controlling row output)

In this chapter, we continue using SELECT to get tabular information out of existing tables, by controlling the rows in the resulting table. The WHERE clause of the SELECT statement will be used to define which rows should be returned. The ORDER BY clause of the SELECT statement will be used to define the order of the rows.

2.3.1 WHERE

2.3.1.1 Basics

Here is a basic SELECT statement to get all first names of people:

```
SELECT firstName
FROM Person
```

If you execute this on the ReferenceDB, the result will be a one-column, many row table. How many rows will there be? There will be one row in the result for every row that was in the Person table at the instant the SELECT was executed. If you execute the same statement again, will you get the same rows back? Only if the table records have not changed at all – that might be quite rare in an OLTP setting. If you execute the same statement again, will you get the same number of rows back? Again, only if no records were added or deleted from the table.

What if we only want to see the first names of certain people from the table. For instance, we only wish to see the first names of women? Here is how to do that:

```
SELECT firstName
FROM Person
WHERE gender='F'
```

Notice that the only change is the addition of the last line, a WHERE clause. The WHERE clause in this case means “only show records where the value of the gender field is equal to a single character, ‘F’”.

One way to think of how the WHERE clause works is to imagine that the database server will actually perform the following steps:

1. Retrieve all records from the Person table
2. Look at each record
   a. if the value of the gender field for this record is equal to ‘F’, keep it
   b. otherwise throw it out
3. Keep only the firstName column out of the remaining records
4. Return what’s left

Of course, the database server may not actually do all of those steps, but it is helpful to think of the WHERE clause in this way as you learn it. So for a record to be included in the result, it must satisfy the condition in the WHERE clause. The condition in the WHERE clause is called a logical expression. So the general format for a SELECT statement now looks like:
A logical expression is any expression that always results in one of two things: True or False. Here are some examples of simple logical expressions that could all be used with a SELECT on the Person table:

- 37 = 55
- SQRT(16) = 4
- gender = 'F'
- weight >= 152.2
- weight/2 <= 45.8
- 'Dr' = salutary

The first example above will obviously always evaluate to False, regardless of the table or record involved. The second example above will always evaluate to True, regardless of the table or record involved. The last four examples all reference a field (in the Person table). Each example could evaluate to True or False, depending on the record that it is currently evaluating. In most cases, the person writing the SELECT statement does not know ahead of time exactly which records will be returned, only that the records will satisfy the WHERE condition. In a dynamic OLTP environment, the records that will satisfy the WHERE condition are changing.

### 2.3.1.2 NOT, AND, and OR

Logical expressions can be combined with other logical expressions with logical operators to create new logical expressions. This makes the WHERE clause quite powerful, but can also make it complex.

The available logical operators are NOT, AND, and OR. The NOT operator takes any logical expression, and negates it; anything that is True becomes False, and vice versa. Here are some examples:

- NOT (37 = 55)
- NOT (SQRT(16) = 4)
- NOT (gender = 'F')
- NOT (weight >= 152.2)
- NOT (weight/2 <= 45.8)
- NOT ('Dr' = salutary)

Any of the above examples can be used in a WHERE clause of a select statement, because they are logical expressions. Notice that these are all exactly the same as the previous set, except that they are
preceded by the NOT operator. The first example (37=55), would normally always evaluate to False, but
the NOT changes it to always evaluating to True. Therefore:

\[ \text{NOT(expression that evaluates True)} \rightarrow \text{False} \]
\[ \text{NOT(expression that evaluates False)} \rightarrow \text{True} \]

NOT is considered a \textit{unary} operator because it operates on a single logical expression. The general
syntax for the NOT operator is:

\[ \text{NOT(logical expression)} \]

The OR operator is a \textit{binary} operator -- it operates on two logical expressions. The syntax for the OR
operator is:

\[ (\text{logical expression}) \text{ OR } (\text{logical expression}) \]

Here are all the possible outcomes for the OR operator:

\[ (\text{True}) \text{ OR } (\text{True}) \rightarrow \text{True} \]
\[ (\text{True}) \text{ OR } (\text{False}) \rightarrow \text{True} \]
\[ (\text{False}) \text{ OR } (\text{True}) \rightarrow \text{True} \]
\[ (\text{False}) \text{ OR } (\text{False}) \rightarrow \text{False} \]

In short, if either of the two logical expressions involved evaluates to True, the result of the OR is True.
Here are some examples, along with how they might evaluate:

\[ (2=3) \text{ OR } (5=5) \rightarrow \text{True} \]
\[ ('Fred'='Fred') \text{ OR } (2=3) \rightarrow \text{True} \]
\[ (\text{SQRT}(4)=2) \text{ OR } (2=2) \rightarrow \text{True} \]
\[ (\text{SQRT}(4)=3) \text{ OR } ('Fred'='George') \rightarrow \text{False} \]
\[ (\text{firstName}='Fred') \text{ OR } (2=2) \rightarrow \text{True} \]
\[ (\text{firstName}='Fred') \text{ OR } (2=3) \rightarrow ? \]

Notice that the evaluation of the last logical expression is unknown, because it depends on the value of
a field in a record. Depending on the value in the record’s firstName field, the logical expression could
evaluate to True or False. Notice also that the next-to-last one always evaluates to True, regardless of
the firstName field value, because the right-hand logical expression is always true.

The AND operator is another binary operator. Here is how the AND operator evaluates the possible
combinations of two logical expressions:

\[ (\text{True}) \text{ AND } (\text{True}) \rightarrow \text{True} \]
\[ (\text{True}) \text{ AND } (\text{False}) \rightarrow \text{False} \]
\[ (\text{False}) \text{ AND } (\text{True}) \rightarrow \text{False} \]
\[ (\text{False}) \text{ AND } (\text{False}) \rightarrow \text{False} \]
For the AND operator to evaluate to True, both logical expressions involved must evaluate to True. Here are some examples, and how they would evaluate

\[(2=3) \text{ AND } (5=5) \rightarrow \text{False}\]

\[('Fred'='Fred') \text{ AND } (2=3) \rightarrow \text{False}\]

\[(\text{SQRT}(4)=2) \text{ AND } (2=2) \rightarrow \text{True}\]

\[(\text{SQRT}(4)=3) \text{ AND } ('Fred'='George') \rightarrow \text{False}\]

\[(\text{firstName}='Fred') \text{ AND } (2=3) \rightarrow \text{False}\]

\[(\text{firstName}='Fred') \text{ AND } (2=2) \rightarrow ?\]

Notice that the last expression is unknown, because it depends on the value of the firstName field. However, the next-to-last one is always False, regardless of the firstName field value, because the right-hand-side is always False.

To see how NOT, OR, and AND are used in practice, here are some SELECT statements, and how they would be translated into and English language description:

```sql
SELECT firstName
FROM Person
WHERE NOT (firstName='Fred')
```

*All first names of people whose first name is not Fred*

```sql
SELECT lastName
FROM Person
WHERE firstName='George'
OR firstName='Fred'
```

*All last names of people whose first name is either George or Fred*

```sql
SELECT firstName
FROM Person
WHERE lastName='Smith'
OR (weight>300 AND gender='M')
```

*All first names of people whose last name is Smith, or who are very heavy males*

```sql
SELECT firstName
FROM Person
WHERE NOT (lastName='Smith' OR (weight>300 AND gender='M'))
```

*All first names of people who were not shown in the last example*

### 2.3.1.3 Functions that return logical values
There are special operators available in SQL that return logical values, and therefore can be used in place of a logical expression. The first that we’ll cover is the IN operator. The general syntax for the IN operator is:

```
expression1 IN (listOfExpressions)
```

To better understand IN, let’s look at an example:

```
SELECT firstName
FROM Person
WHERE lastName IN ('Smith', 'Jones')
```

*The first names of people whose last name is either Smith or Jones.*

Note that the above SELECT statement can be written equivalently as:

```
SELECT firstName
FROM Person
WHERE (lastName='Smith')
OR (lastName='Jones')
```

Strictly speaking, the IN operator is not necessary – any IN expression can be written as a sequence or OR expressions. However, it can be a very good “shorthand” to make the SQL statement more readable. If the list of lastNames had 15 entries, the OR phrasing would be very long compared to the IN phrasing. Also, the IN operator can be combined with NOT in this way:

```
SELECT firstName
FROM Person
WHERE NOT (lastName IN ('Smith', 'Jones'))
```

*The first names of people whose last name is neither Smith nor Jones.*

These last two become much harder to phrase with only NOT, AND, and OR:

```
SELECT firstName
FROM Person
WHERE NOT ((lastName='Smith') AND (lastName='Jones'))
```

There is another way to use the IN clause that we’ll cover later with subqueries – the *expressionList* can actually be the result of a SELECT statement. In other words, the *expressionList* is not known until the query is run. This is a very powerful capability that is difficult to accomplish without the IN operator.

Another logical operator that is important when selecting records based on string values is the LIKE operator. The LIKE operator gives string matching capability based on wildcards to the WHERE clause. The general syntax of the LIKE operator is:

```
enExpression LIKE pattern
```
stringExpression LIKE formatExpression

where stringExpression is any expression that results in a string of characters

and formatExpression is a special string of characters that defines the matching condition

Before we cover the details, here are several examples to get the feel of the LIKE operator:

```
SELECT firstName
FROM   Person
WHERE  lastName LIKE 'K%'

The first names of all people whose last name begins with the letter K
```

```
SELECT firstName
FROM   Person
WHERE  lastName LIKE '%K'

The first names of all people whose last name ends with the letter K
```

```
SELECT firstName
FROM   Person
WHERE  lastName LIKE '_K%'

The first names of all people whose last name has K as the second letter
```

From the above examples, you may already see what the _ and % wildcard characters mean:

- % matches on zero or more characters
- _ matches on exactly one character

Here are various logical expressions using LIKE, and what they evaluate to:

- 'Smith' LIKE 'S%' → True
- 'Smith' LIKE '%' → True
- 'Smith' LIKE '_' → False
- 'Smith' LIKE '_m%' → True
- 'Smith' LIKE '__i%' → True
- 'Smith' LIKE 'S%h' → True
- 'Smith' LIKE '%t%' → True
- 'Smith' LIKE '_m%t_' → True

The above examples are good to see the exact behavior of LIKE, but may not be realistic in a work setting. Here are some examples that show more typical realistic usage:
SELECT *
FROM Company
WHERE companyName LIKE 'Mi%'

All companies whose names begin with ‘Mi’

SELECT productID
FROM Product
WHERE productDescription LIKE '%stainless%'

All products that have the word stainless in their description, (i.e., all the stainless steel products)

SELECT firstName,
      lastName
FROM Person
WHERE (firstName LIKE 'D%')
      AND (lastName LIKE 'Mc%')

All people whose first name starts with D and lastname starts with Mc (i.e., I can’t remember that person’s name but it’s something like this...)

The WHERE clause is used to accomplish filtering of records, which is an important way to create useful information from raw data. In most organizations today there is no lack of data. In fact, there is generally too much data, which makes it difficult to see patterns or make decisions. By filtering out data that does not apply to the current task, you increase the value of the information. It is common to filter data by time period to see only recent data, or data for a certain time period, e.g., sales for last quarter. It is also common to use filtering to drill-down into data, moving from aggregate data to detailed data. For example, sales data might first be displayed by year, then by clicking on a certain year, all sales data from other years are filtered. Then the data is shown within that year by quarter. By clicking on a certain quarter, the other quarters are filtered, and the sales might then be shown by month, etc.

2.3.2 Dates

2.3.2.1 Date data types

Dates are important pieces of data that are necessary to organizations. However, dates have inherent oddities that make them harder to deal with than mere numbers or characters. Here are some challenges:

- Months have different numbers of days
- Each month can begin on a different day of the week
- Years can have different numbers of days (leap years)
- Dates can be formatted in many different ways
- Dates can be stored with various precisions (to the day, hour, year, etc.)
- Special days (holidays, birthdays, etc.)

These oddities make some common questions about dates very difficult to answer:

- How many days between two dates?
- How many work days between two dates?
- How old (in years) is a person on a specific date?

Because of the inherent difficulties of dealing with dates, database engines have special data types for dates, and the SQL language includes special functions to help with date processing.

In SQL Server 2005, the main data type for dates is the datetime (in SQL Server 2008, consider using the new data type, datetime2.) As its name suggests, datetime stores both date and time information. The datetime data type can store dates between January 1, 1753 and December 31, 9999 to an accuracy of 3.33 milliseconds. In addition, there are functions specific to datetimes that perform commonly needed functions. Here are a sampling:

- GETDATE() – returns the current server datetime
- DAY(datetime) – returns the day of the month, as a whole number
- MONTH(datetime) – returns the month, as a whole number
- YEAR(datetime) – returns the year, as a whole number
- DATEADD(datepart, number, date
time) – returns a new datetime by adding the specified number of dateparts to the given datetime
- DATEDIFF(datepart, startdate, enddate) – returns the number of dateparts between the startdate and enddate

There are other datetime functions, but this will covers the basics. Here is a simple example that gives the date parts of the current date and time:

```
SELECT MONTH(GETDATE()) AS [month],
       DAY(GETDATE()) AS [day of the month],
       YEAR(GETDATE()) AS [year]
```

The next example shows how to add 432 weeks to a specific date:

```
SELECT DATEADD(week, 432, '10,2,1998') AS [future datetime]
```

Here is how to add 37 hours to the current datetime:

```
SELECT DATEADD(hour, 37, GETDATE()) AS [future datetime]
```

Now, subtract 4 years from the current datetime:

```
SELECT DATEADD(year, -4, GETDATE()) AS [past datetime]
```
In the examples above, the first parameter of the DATEADD() function specifies the units to be added (subtracted) from the datetime. You can specify year, quarter, month, week, day, hour, minute, second, or millisecond.

Here is an example of how to find the difference, in various units, between two dates:

```sql
SELECT DATEDIFF(year, '3/5/1972', GETDATE()) AS [years],
       DATEDIFF(days, '3/5/1972', GETDATE()) AS [days],
       DATEDIFF(hours, '3/5/1972', GETDATE()) AS [hours]
```

Note that in the example above, leap years are dealt with accurately. You can check this by running the following statement:

```sql
SELECT DATEDIFF(year, '3/5/1972', GETDATE())*365 AS [daysA],
       DATEDIFF(days, '3/5/1972', GETDATE()) AS [daysB]
```

In short, you are best advised to learn and use the date and time data types on the database that you are using. This will save you time and produce more accurate information when dealing with dates and times.

In addition to the datetime functions available to you, comparison operators such as equal (=), not equal (<>), greater than (>), etc., will work on datetimes. Also, there is a special logical operator that is especially useful for datetimes. Here is an example:

```sql
SELECT ID, orderAmount
FROM Sale
WHERE orderDate BETWEEN '1/1/2001' AND '1/31/2001'
```

This returns all Sale records for the month of January 2001. Note that this is inclusive – orders on Jan 1 and Jan 31 are included. To exclude Jan 1 and Jan 31, this would need to be rephrased as:

```sql
SELECT ID, orderAmount
FROM Sale
WHERE orderDate > '1/1/2001'
     AND orderDate < '1/31/2001'
```

The BETWEEN...AND operator can be used for numbers and characters also, but it is especially useful for dates.

It should be noted that there is an implicit conversion that occurs when dates are specified with quotes. For example, ‘1/1/2001’ is a set of characters (a string) that is converted, by the database engine, into a datetime data type. This is called implicit, because we are not explicitly specifying what data type we intend. To be explicit, we would need to use the CONVERT() or CAST() functions to specify the exact data type. At this stage, implicit conversion is simple and suits our needs for ad-hoc, throw-away SQL. However, for SQL that will be embedded in systems, it pays to be explicit.
2.3.3 ORDER BY

The WHERE clause gives the ability to control which rows are returned by a select statement, but it doesn’t control the order of the rows. And even though the ordering of the rows seems that it follows a pattern, or is somewhat predictable, it is not dependable. There is no order guaranteed in the result of a SELECT statement, unless there is an ORDER BY clause.

The ORDER BY clause is (generally) the last clause in a SELECT statement. Here is a simple SELECT statement with an ORDER BY:

```
SELECT firstName, lastName
FROM Person
ORDER BY lastName ASC
```

In this statement, the returned rows will be sorted alphabetically by lastName. The ASC keyword is used to sort ascending, or lowest to highest; in the alphabetic sense, from A to Z. To sort descending, the keyword DESC is used in place of ASC. If neither ASC nor DESC is specified, the default sorting is ASC.

The value used to order the records can be an expression, rather than a simple field name, as in this example:

```
SELECT lastName, weight/14 AS [weight in stones]
FROM Person
ORDER BY weight/14 ASC
```

Here, the Person records are order by the British unit of weight measure, stones. Of course, the expression could be quite complicated. Anything that might be in a SELECT clause expression can be used in the ORDER BY clause. In fact, if an expression in the SELECT clause has been given a column name with the AS statement, the column name can be used in the ORDER BY clause:

```
SELECT lastName, firstName, weight
FROM Person
WHERE lastName LIKE 'J%'
ORDER BY lastName ASC, weight DESC
```

The general form of the ORDER BY clause is:

```
ORDER BY expression [ASC|DESC], expression [ASC|DESC],...
```

Each expression in the ORDER BY is called a sort key. The sort keys take precedence from left to right.
Here is a possible set of records that could be returned from the above SELECT statement:

<table>
<thead>
<tr>
<th>lastName</th>
<th>firstName</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jeffries</td>
<td>Jill</td>
<td>135</td>
</tr>
<tr>
<td>Jenkins</td>
<td>Joseph</td>
<td>187</td>
</tr>
<tr>
<td>Jones</td>
<td>William</td>
<td>256</td>
</tr>
<tr>
<td>Jones</td>
<td>David</td>
<td>225</td>
</tr>
<tr>
<td>Jones</td>
<td>Robert</td>
<td>191</td>
</tr>
<tr>
<td>Jones</td>
<td>Janet</td>
<td>129</td>
</tr>
</tbody>
</table>

There are several things to note about the results. First, looking at each column individually, the records appear to only be sorted by lastName, ascending, even though the weight column is specified as a sort key. That’s because the lastName sort key was specified first in the list of sort keys. Second, looking at the weight column within the group of records that all have Jones as the lastName (the last 4 records), the records are sorted, descending. In short, the weight sort key is only meaningful in groups of records with equivalent lastName sort key values.

Sorting is a very important operation used to create high value information from raw data, and it is common to use multiple sort keys to create information for a specific purpose. As an example, suppose a retailer wished to determine the higher priced items in each product category. Here is a SELECT statement that might be helpful:

```sql
SELECT categoryName, productName, retailPrice
FROM PRODUCT
ORDER BY categoryName ASC, retailPrice DESC
```

2.3.4 Formatting

With what has been covered to this point, some of the SELECT statements can become quite lengthy, and difficult to read and understand. To aid in readability and make them more easy to understand, it is best to have a standard way of formatting them. These are general guidelines that don’t necessarily apply to all situations, but will generally help in readability.

- SQL reserved words such as SELECT, FROM, WHERE, AND, etc. should be capitalized
- SELECT should begin on its own line
- If the SELECT statement goes onto the next line(s), indent the next line(s)
- FROM should begin on its own line
- WHERE should begin on its own line
- If the WHERE statement goes onto the next line(s), indent the next line(s)
- Each logical expression in the WHERE clause should be on its own line
Here is an example of a nicely formatted, complex SELECT statement:

```sql
SELECT firstName, lastName, middleInitial, salutary, weight, dateOfBirth
FROM Person
WHERE (lastName LIKE 'K%')
    AND (weight > 150)
    AND (firstName LIKE '%t%')
```

An even more nicely formatted SELECT statement is:

```sql
SELECT firstName AS [first name],
    lastName AS [last name],
    middleInitial AS [middle initial],
    salutary AS [salutary],
    weight AS [weight in pounds],
    dateOfBirth AS [date of birth]
FROM Person
WHERE (lastName LIKE 'K%')
    AND (firstName LIKE '%t%')
    AND (weight > 150)
```

One way to think of your SQL statements is as a way to communicate to the database engine the set of which records you’d like. That is certainly true. In addition, you’d like to communicate to other people which set of records you’d like. If you can do both, it will save time in debugging, maintenance, and new feature additions.