CHAPTER 11

# The SELECT Statement

You will use the `SELECT` statement more than any other T-SQL statement in your database career. This is an axiom that applies to DBAs as much as it does to developers. `SELECT` is your friend, your assistant, your sidekick. Whenever you have a question about your database, chances are `SELECT` will give you the answer. That said, `SELECT` in the wrong hands can be evil. Evil! It's easier to make `SELECT` give you the wrong data than the right data (much easier, usually). Let's see if we can keep `SELECT` on the side of the good guys.

## What Is the SELECT Statement For?

The `SELECT` statement can be used for a number of purposes, but its main aim is to return a set of data. A *set* is a collection of rows, and in SQL, thinking in sets is very important. Many developers focus on working on one row of data at a time, but SQL Server makes it very easy to work on sets of data, meaning you can insert, update, and delete multiple records with one statement, rather than processing one record at a time. I'll talk about this more in Chapter 19. The `SELECT` statement is very important when dealing with sets, as it usually the `SELECT` statement that generates the set of data for you.

`SELECT` can also be used to modify rows, in conjunction with the aforementioned `INSERT INTO`, `UPDATE`, and `DELETE` statements.

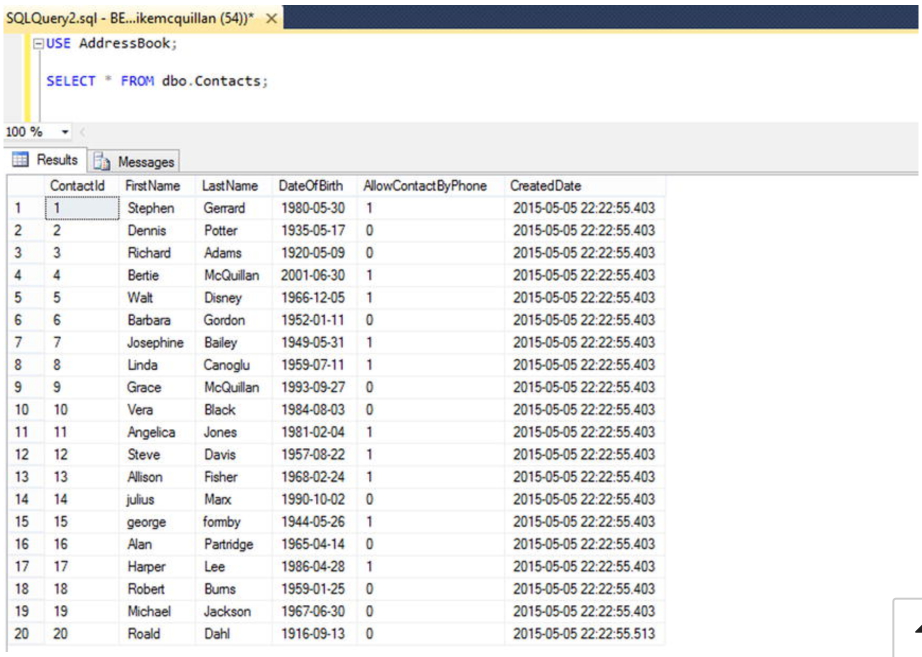
## The Simple SELECT Statement

Let's begin by taking a look at the simplest form of `SELECT` statement. We'll write a `SELECT` that shows all records in the `Contacts` table.

```
USE AddressBook;

SELECT * FROM dbo.Contacts;
```

If you rebuilt your database at the end of the last chapter, this should return the 20 rows displayed in Figure 11-1.



```
USE AddressBook;

SELECT * FROM dbo.Contacts;
```

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 2 | Dennis | Potter | 1935-05-17 | 0 | 2015-05-05 22:22:55.403 |
| 3 | 3 | Richard | Adams | 1920-05-09 | 0 | 2015-05-05 22:22:55.403 |
| 4 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 5 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-05-05 22:22:55.403 |
| 6 | 6 | Barbara | Gordon | 1952-01-11 | 0 | 2015-05-05 22:22:55.403 |
| 7 | 7 | Josephine | Bailey | 1949-05-31 | 1 | 2015-05-05 22:22:55.403 |
| 8 | 8 | Linda | Canoglu | 1959-07-11 | 1 | 2015-05-05 22:22:55.403 |
| 9 | 9 | Grace | McQuillan | 1993-09-27 | 0 | 2015-05-05 22:22:55.403 |
| 10 | 10 | Vera | Black | 1984-08-03 | 0 | 2015-05-05 22:22:55.403 |
| 11 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 12 | 12 | Steve | Davis | 1957-08-22 | 1 | 2015-05-05 22:22:55.403 |
| 13 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 14 | 14 | julius | Marx | 1990-10-02 | 0 | 2015-05-05 22:22:55.403 |
| 15 | 15 | george | formby | 1944-05-26 | 1 | 2015-05-05 22:22:55.403 |
| 16 | 16 | Alan | Partridge | 1965-04-14 | 0 | 2015-05-05 22:22:55.403 |
| 17 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-05-05 22:22:55.403 |
| 18 | 18 | Robert | Burns | 1959-01-25 | 0 | 2015-05-05 22:22:55.403 |
| 19 | 19 | Michael | Jackson | 1967-06-30 | 0 | 2015-05-05 22:22:55.403 |
| 20 | 20 | Roald | Dahl | 1916-09-13 | 0 | 2015-05-05 22:22:55.513 |

***Figure 11-1.*** *A simple SELECT statement*

This statement is about as simple as it can be. `SELECT *` tells SQL Server to return every column in whatever table (or tables) we specify. The `*` is a shorthand way of saying "give me every column." The `*` is actually very bad and we'll be ditching it shortly!

`FROM` tells SQL Server that we are about to give it the names of the tables from which we want data. We write `FROM dbo.Contacts`, so this `SELECT` statement is saying to display every column for every row in the `Contacts` table—which is exactly what we can see in Figure 11-1.

**WHY IS * A BAD THING?**

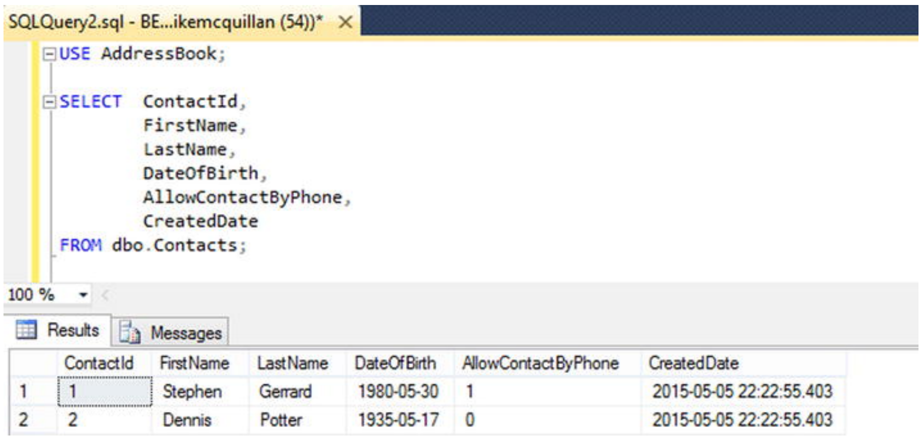
`SELECT *` won't usually pass the steely gaze of a weatherworn DBA. This is because it's good practice only to return what you need from your `SELECT` statements, and also because it isn't explicit about what your `SELECT` statement is doing. If you always return every column, you could be sending lots of extra data across the network whenever a program calls your `SELECT` statement. Always be explicit—we'll be (mostly!) explicit from this point onward.

We are going to write a `SELECT` statement that only lists those contacts who agreed we could contact them via their phone number. Before we do that, we should implement best practice and remove the `*` from our statement, replacing it with the names of the columns we are interested in.

```
USE AddressBook;

SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate
FROM dbo.Contacts;
```

Press F5 to run this; the results (partly shown in Figure 11-2) will be exactly the same.



```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
FROM dbo.Contacts;
```

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 2 | Dennis | Potter | 1935-05-17 | 0 | 2015-05-05 22:22:55.403 |

***Figure 11-2.*** *Explicitly declaring columns with a `SELECT` statement*

Obviously, `*` can be used if you are just writing a quick `SELECT` statement for your own use, or to inspect a table for some reason or other. Just make sure you never use it in production code. Always think of the next person!

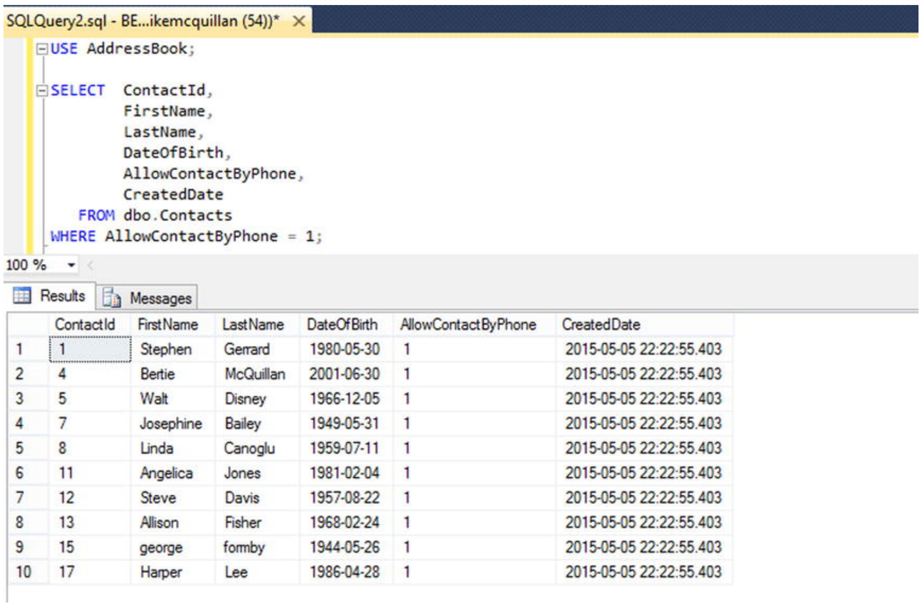
### The WHERE Clause

It is possible to filter the results of the `SELECT` statement in a variety of ways, but the most common method is to use the `WHERE` clause. This allows you to tell SQL Server only to return rows that meet a particular set of criteria. The criteria must use columns that are present in the result set returned by the tables in the `SELECT` statement. Our current `SELECT` statement only uses the `Contacts` table, so we could filter based on any of those columns.

We want to filter on the `AllowContactByPhone` column. We are looking for records where the value of that column is set to 1. In the following code, add the `WHERE` line in bold:

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Co
```

This is the same `SELECT` statement we wrote earlier, except we've added a `WHERE` clause to the end. The `WHERE` clause is saying only to return records where the `AllowContactByPhone` column's value is set to `1`. Exactly 10 of our contacts have this particular value in that particular column. You can see these contacts in Figure 11-3.



SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
   FROM dbo.Contacts
WHERE AllowContactByPhone = 1;
```

100 %

Results | Messages

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 3 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-05-05 22:22:55.403 |
| 4 | 7 | Josephine | Bailey | 1949-05-31 | 1 | 2015-05-05 22:22:55.403 |
| 5 | 8 | Linda | Canoglu | 1959-07-11 | 1 | 2015-05-05 22:22:55.403 |
| 6 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 7 | 12 | Steve | Davis | 1957-08-22 | 1 | 2015-05-05 22:22:55.403 |
| 8 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 9 | 15 | george | formby | 1944-05-26 | 1 | 2015-05-05 22:22:55.403 |
| 10 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-05-05 22:22:55.403 |

***Figure 11-3.*** *A SELECT statement using a WHERE clause*

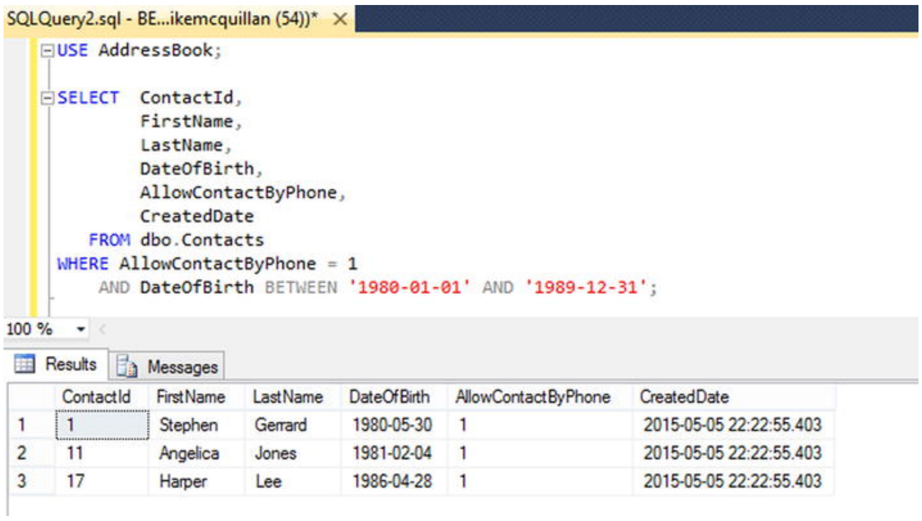
Look carefully at the `AllowContactByPhone` column; all values are set to `1`. We can extend the `WHERE` clause to specify multiple conditions—for instance, we might say give me contacts whom I can phone who were born in the 1980s. (I want to talk to them about Hanson! Look them up on Wikipedia.) There is absolutely no limit to the number of clauses you can add to a `WHERE` clause; you can specify as many as you like.

The AND Operator

We'll use the `AND` operator to add our 1980s clause:

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate
FROM dbo.Contacts
WHERE AllowContactByPhone = 1
AND DateOfBirth BETWEEN '1980-01-01' AND '1989-12-31';
```

The `AND` operator is saying that both condition 1 (allowing contact by phone) and condition 2 (the date of birth should be in the 1980s) must be met for a row to be returned. As you can see in Figure 11-4, three records match our `WHERE` clause.


SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts
WHERE AllowContactByPhone = 1
    AND DateOfBirth BETWEEN '1980-01-01' AND '1989-12-31';
```

100 %

Results | Messages

|   | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 3 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-05-05 22:22:55.403 |

***Figure 11-4.*** *Enhanced `SELECT` statement using `AND`*

We used the `BETWEEN` operator to specify a range that our criteria had to meet—in this instance, a date range. A *range* tells SQL Server that you are looking for records that have a value that falls on or between the values you've specified. We entered the first and the last dates of the 1980s, so any record with a date of birth on or between those would have been picked up.

There are also some other useful operators to be aware of, as shown in Table 11-1.

***Table 11-1.*** *T-SQL Operators*

| **Operator** | **Description** |
|---|---|
| `>` | Greater than |
| `>=` | Greater than or equal to |
| `=` | Equal to |
| `<` | Less than |
| `<=` | Less than or equal to |
| `!= or <>` | Not equal to |
| `IN (values)` | Required value must meet one of the specified values |

| Operator | Description |
|---|---|
| `BETWEEN Value1 AND Value2` | Value must fall on or between the two specified values |

`BETWEEN` is the better option in our example (simply because it is easier to read and maintain), but it also could have been written as:

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

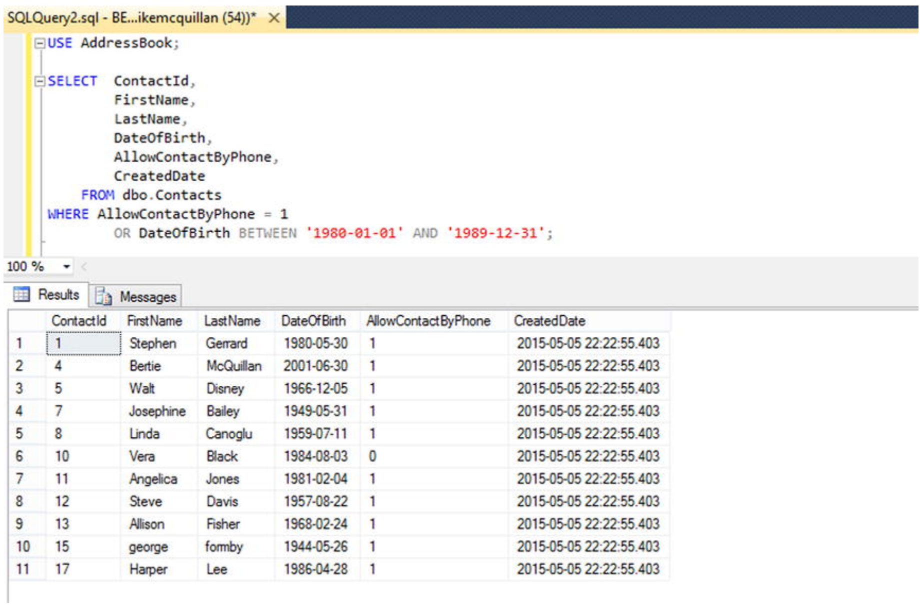
The results are the same, but the query is not as elegant—we've had to specifically state that both the >= and <= apply to the `DateOfBirth` column. We only needed to specify the column name once using `BETWEEN`.

The OR Operator

Now, we've decided that we want to change the query to return all contacts who have either allowed contact by phone or who were born in the 1980s. To be clear: if the contact was born in the 1980s, it doesn't matter whether we can contact them by phone or not—they should be returned. If the contact was not born in the 1980s, they should only be returned if they can be contacted by telephone. The `AND BETWEEN` statement we wrote earlier will not suffice here, so we change the `AND` to `OR`.

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

This is the same statement we wrote earlier; we just swapped out `AND` for `OR`. The results, as you can see in **Figure 11-5**, are quite different.


SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts
WHERE AllowContactByPhone = 1
        OR DateOfBirth BETWEEN '1980-01-01' AND '1989-12-31';
```

100 %

Results | Messages

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 3 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-05-05 22:22:55.403 |
| 4 | 7 | Josephine | Bailey | 1949-05-31 | 1 | 2015-05-05 22:22:55.403 |
| 5 | 8 | Linda | Canoglu | 1959-07-11 | 1 | 2015-05-05 22:22:55.403 |
| 6 | 10 | Vera | Black | 1984-08-03 | 0 | 2015-05-05 22:22:55.403 |
| 7 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 8 | 12 | Steve | Davis | 1957-08-22 | 1 | 2015-05-05 22:22:55.403 |
| 9 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 10 | 15 | george | formby | 1944-05-26 | 1 | 2015-05-05 22:22:55.403 |
| 11 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-05-05 22:22:55.403 |

***Figure 11-5.*** *Using the OR operator*

Eleven rows are now returned. Ten of them we've already seen—they match the first criteria regarding `AllowContactByPhone = 1`. Row 6 contains a record that has `AllowContactByPhone` set to `0`. It doesn match the first criteria but it does match the second, as the contact was born in the 1980s.

What we were saying when we were using AND was:

- Find me contacts that allow contact by phone, and who were born in the 1980s

What we are now saying with OR is:

- Find me contacts that either allow contact by phone, or were born in the 1980s

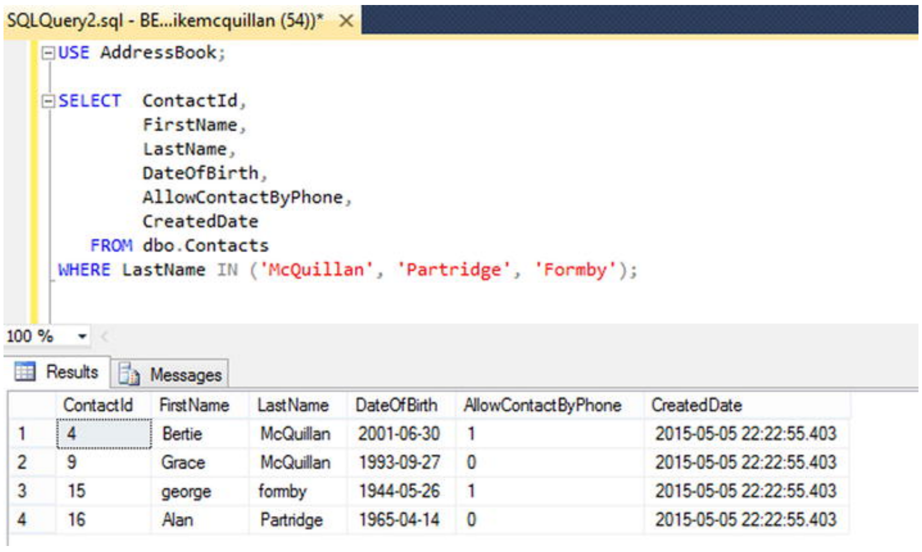
With the AND operator, all of the criteria had to be matched. With the OR operator, any one of the criteria needs a match. This is an important distinction.

### The IN Operator

BETWEEN allows you to filter using a range of values, but IN allows you to filter using specific values. Say we only want to find contacts whose surname is one of McQuillan, Partridge, or Formby.

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

We tell the IN operator which values we are interested in by specifying them in the parentheses. You can place as many values as you want in the parentheses. Running it brings back the four expected results (shown in Figure 11-6).



SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts
WHERE LastName IN ('McQuillan', 'Partridge', 'Formby');
```

|  | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 9 | Grace | McQuillan | 1993-09-27 | 0 | 2015-05-05 22:22:55.403 |
| 3 | 15 | george | formby | 1944-05-26 | 1 | 2015-05-05 22:22:55.403 |
| 4 | 16 | Alan | Partridge | 1965-04-14 | 0 | 2015-05-05 22:22:55.403 |

***Figure 11-6.*** *Using IN to filter*

This statement is equivalent to specifying two OR operators instead:

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

I'm sure you'll agree that the IN version is much easier to read and extend if necessary!

### Mixing Operators

We've covered a huge amount of ground already, yet we've hardly scratched the surface of the SELECT statement's capabilities. Let's look at something a little more advanced. We've seen how to use operators like AND, OR, and IN. But we can actually mix and match these in the same statement should we wish.

**USE PARENTHESES TO GROUP CONDITIONS**

When specifying multiple conditions that use OR statements, I recommend using parentheses to logically gro them. I've often been stumped with OR statements when I haven't used parentheses!

Here's a more complicated WHERE clause.

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

Note there are two parentheses at the end there. This returns 13 rows, broken down as follows:

- Four rows match the first criteria: allow contact by phone and born in 1970 or after.
- Seven rows match the second criteria: do not allow contact by phone and born before 1970.
- The last two rows match the surnames specified in the final clause.

Note that we've linked two conditions together using AND, and enclosed those in parentheses. The enclosure makes it clear what the statement is trying to do. If the parentheses were not there, the statement would be much harder to read—try taking them out and see what you think.

This statement will work with or without the parentheses, but quite often not using them will produce misleading results, so be very careful. I always add them when I'm using OR; it's good practice.

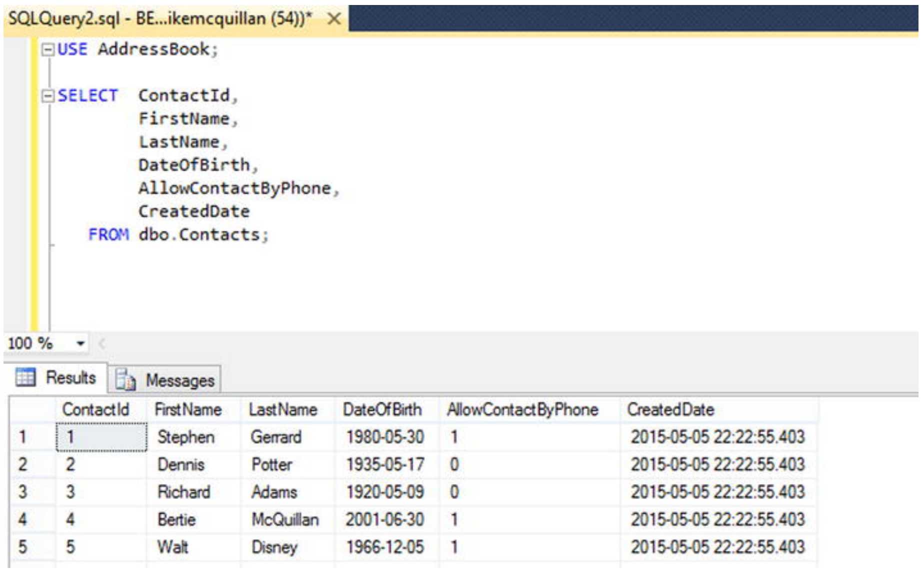
**THE YEAR() FUNCTION**

The preceding example uses the built-in T-SQL function YEAR(). This accepts a date (which comes from the DateOfBirth column in the example) and returns the year specified in that date. So a date of 1974-10-03 would return 1974. There are many date functions in SQL Server, including the obvious MONTH() and DAY(). Take a look at what's on offer—there's some powerful stuff available.

More information on system functions can be found in Appendix C.

## Ordering Data

We're pretty conversant with the SELECT statement now. We can pull the data from a table and filter it in many wild and wonderful ways. Now we need to figure out how to order that data. If we run a SELECT statement without any filters, the records in the Contacts table are returned ordered by the ContactId column. Figure 11-7 proves this:



SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts;
```

100 %

Results | Messages

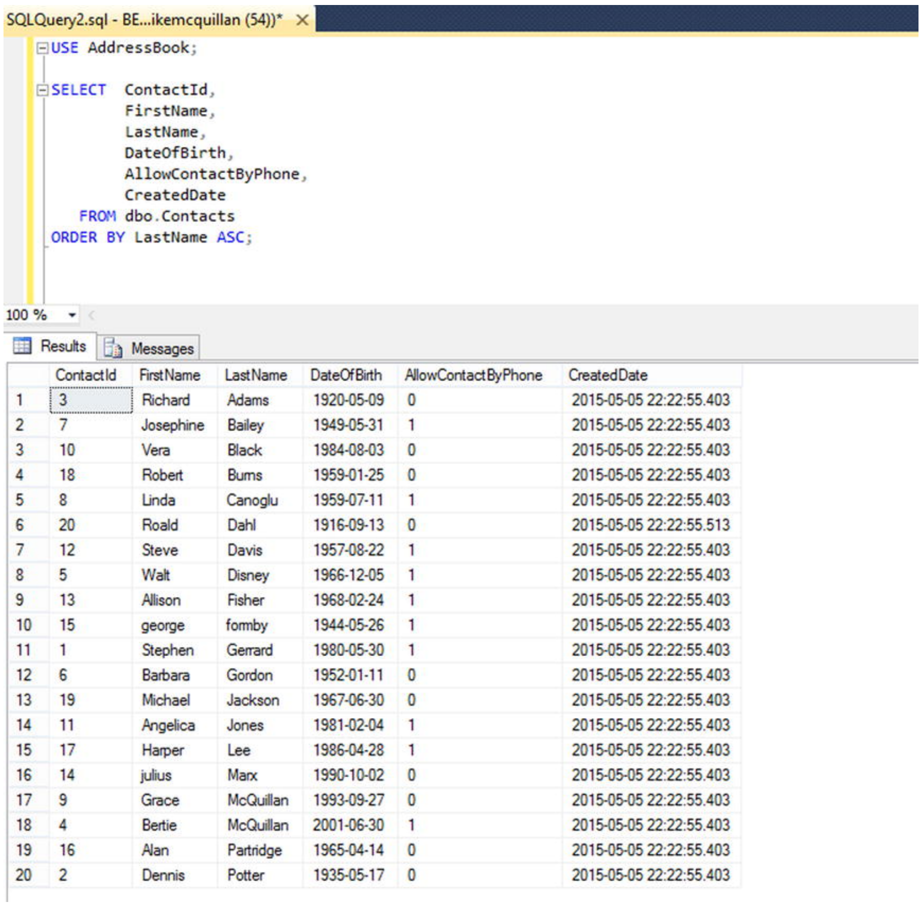
| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 2 | 2 | Dennis | Potter | 1935-05-17 | 0 | 2015-05-05 22:22:55.403 |
| 3 | 3 | Richard | Adams | 1920-05-09 | 0 | 2015-05-05 22:22:55.403 |
| 4 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 5 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-05-05 22:22:55.403 |

***Figure 11-7.*** *A naturally ordered SELECT statement*

We can actually order by any column or combination of columns we choose, using the ORDER BY clause. We'll modify the SELECT statement to order by the LastName column, in alphabetical order.

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

`ASC` is short for ascending. This is actually the default and we could have omitted it, but it's always best to be explicit—remember, think about the next person! Lo and behold, by the magic of SQL Server, our results are returned, sorted by the `LastName` column. Check out Figure 11-8.



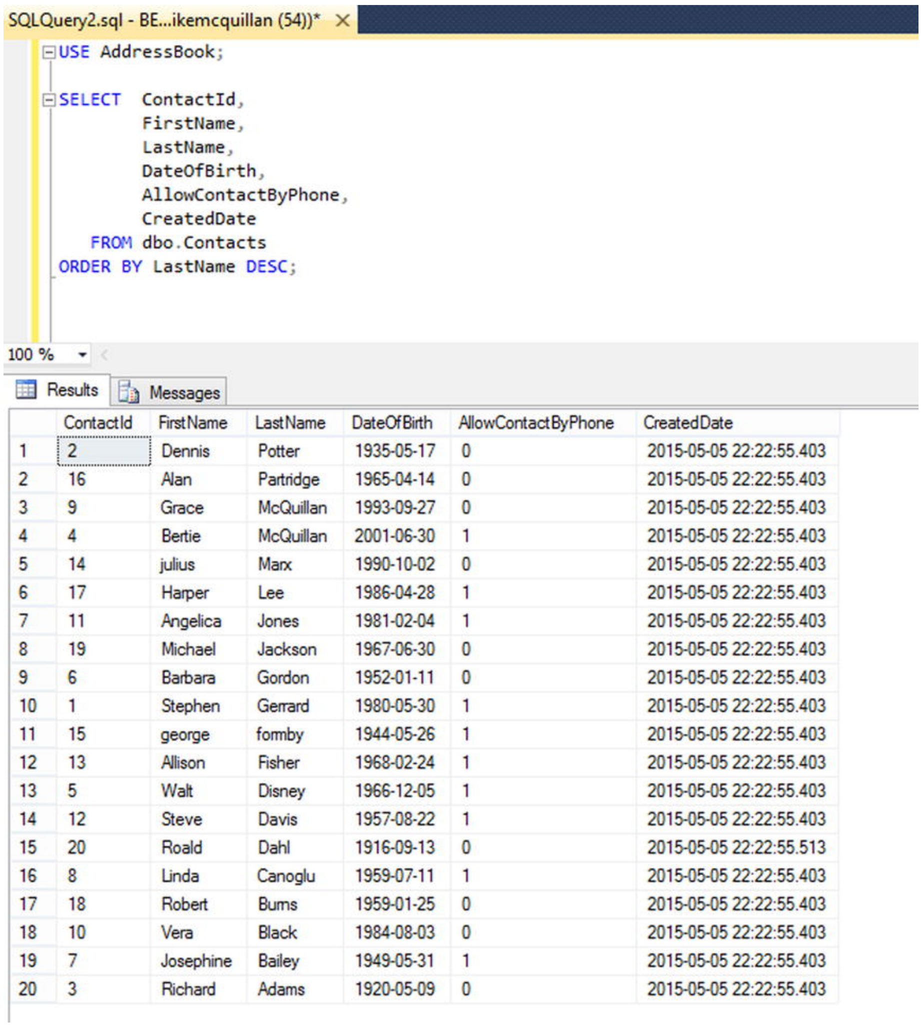
```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts
ORDER BY LastName ASC;
```

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 3 | Richard | Adams | 1920-05-09 | 0 | 2015-05-05 22:22:55.403 |
| 2 | 7 | Josephine | Bailey | 1949-05-31 | 1 | 2015-05-05 22:22:55.403 |
| 3 | 10 | Vera | Black | 1984-08-03 | 0 | 2015-05-05 22:22:55.403 |
| 4 | 18 | Robert | Burns | 1959-01-25 | 0 | 2015-05-05 22:22:55.403 |
| 5 | 8 | Linda | Canoglu | 1959-07-11 | 1 | 2015-05-05 22:22:55.403 |
| 6 | 20 | Roald | Dahl | 1916-09-13 | 0 | 2015-05-05 22:22:55.513 |
| 7 | 12 | Steve | Davis | 1957-08-22 | 1 | 2015-05-05 22:22:55.403 |
| 8 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-05-05 22:22:55.403 |
| 9 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 10 | 15 | george | formby | 1944-05-26 | 1 | 2015-05-05 22:22:55.403 |
| 11 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 12 | 6 | Barbara | Gordon | 1952-01-11 | 0 | 2015-05-05 22:22:55.403 |
| 13 | 19 | Michael | Jackson | 1967-06-30 | 0 | 2015-05-05 22:22:55.403 |
| 14 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 15 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-05-05 22:22:55.403 |
| 16 | 14 | julius | Marx | 1990-10-02 | 0 | 2015-05-05 22:22:55.403 |
| 17 | 9 | Grace | McQuillan | 1993-09-27 | 0 | 2015-05-05 22:22:55.403 |
| 18 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 19 | 16 | Alan | Partridge | 1965-04-14 | 0 | 2015-05-05 22:22:55.403 |
| 20 | 2 | Dennis | Potter | 1935-05-17 | 0 | 2015-05-05 22:22:55.403 |

***Figure 11-8.*** *A `SELECT` statement in ascending order using `ORDER BY`*

Not bad, eh? Now change `ASC` to `DESC`. You've guessed it—this will reverse the order, sorting by `LastName` using Z–A ordering. You can see the reversed rows in Figure 11-9.



```
USE AddressBook;

SELECT  ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
   FROM dbo.Contacts
ORDER BY LastName DESC;
```

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 2 | Dennis | Potter | 1935-05-17 | 0 | 2015-05-05 22:22:55.403 |
| 2 | 16 | Alan | Partridge | 1965-04-14 | 0 | 2015-05-05 22:22:55.403 |
| 3 | 9 | Grace | McQuillan | 1993-09-27 | 0 | 2015-05-05 22:22:55.403 |
| 4 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |
| 5 | 14 | julius | Marx | 1990-10-02 | 0 | 2015-05-05 22:22:55.403 |
| 6 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-05-05 22:22:55.403 |
| 7 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 8 | 19 | Michael | Jackson | 1967-06-30 | 0 | 2015-05-05 22:22:55.403 |
| 9 | 6 | Barbara | Gordon | 1952-01-11 | 0 | 2015-05-05 22:22:55.403 |
| 10 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-05-05 22:22:55.403 |
| 11 | 15 | george | formby | 1944-05-26 | 1 | 2015-05-05 22:22:55.403 |
| 12 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 13 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-05-05 22:22:55.403 |
| 14 | 12 | Steve | Davis | 1957-08-22 | 1 | 2015-05-05 22:22:55.403 |
| 15 | 20 | Roald | Dahl | 1916-09-13 | 0 | 2015-05-05 22:22:55.513 |
| 16 | 8 | Linda | Canoglu | 1959-07-11 | 1 | 2015-05-05 22:22:55.403 |
| 17 | 18 | Robert | Burns | 1959-01-25 | 0 | 2015-05-05 22:22:55.403 |
| 18 | 10 | Vera | Black | 1984-08-03 | 0 | 2015-05-05 22:22:55.403 |
| 19 | 7 | Josephine | Bailey | 1949-05-31 | 1 | 2015-05-05 22:22:55.403 |
| 20 | 3 | Richard | Adams | 1920-05-09 | 0 | 2015-05-05 22:22:55.403 |

***Figure 11-9.*** *A `SELECT` statement in descending order using `ORDER BY`*

You can sort one column in ascending order and another in descending order, as in this example:

```
SELECT ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM dbo.Conta
```

If you change the `ContactId DESC` to `ContactId ASC`, you'll notice the order of the two records change.

It's often wise to specify an `ORDER BY` clause in your `SELECT` statements. It explicitly tells the database how you want the data to be returned, and it guarantees your `SELECT` statement will always be consistent in how it returns data.

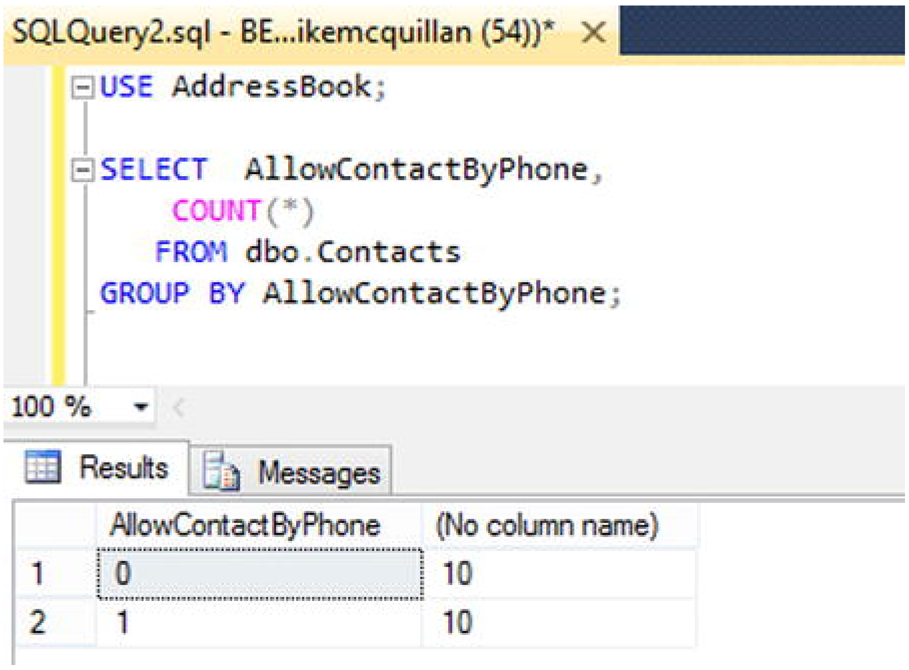
**Grouping Data**

We're motoring along nicely, and it is time for a quick look at how the `GROUP BY` clause can aggregate data for us. With `GROUP BY`, we can break our data down into groups, and a row will be returned for each group displaying whatever information we have requested. `GROUP BY` is usually used to obtain some kind of statistical information, such as the number of records that match a particular type.

Let's say we wanted to know how many contacts we had that allow phone calls, and how many that don't. We can write:

```
SELECT AllowContactByPhone, COUNT(*) FROM dbo.Contacts GROUP BY AllowContactByPhone;
```

`COUNT()` is an aggregate function, one of many provided as part of the T-SQL language. `COUNT()` will return the number of rows matching the criteria specified by the `GROUP BY` clause, so in this case it will return a count for each different value of `AllowContactByPhone`. There are only two values in the table for this column: `1` and `0`. So we can expect two rows to be returned (Figure 11-10).



SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  AllowContactByPhone,
    COUNT(*)
    FROM dbo.Contacts
GROUP BY AllowContactByPhone;
```

100 %

Results | Messages

|   | AllowContactByPhone | (No column name) |
|---|---|---|
| 1 | 0 | 10 |
| 2 | 1 | 10 |

***Figure 11-10.*** *A* `GROUP BY` *using* `COUNT()`

There are 10 records of each type in the `Contacts` table. You can group on more than one column; the only rule is whatever columns appear in the `SELECT...FROM` must also appear in the `GROUP BY` clause.

**AGGREGATE FUNCTIONS**

T-SQL provides a number of aggregate functions that can greatly enhance your queries. Many of these are designed to work with or without a `GROUP BY` clause.

You can find a list of all the aggregate functions at `https://msdn.microsoft.com/en-us/library/ms173454.aspx`. Some of these are outlined in Appendix C.

The `COUNT()` function can be used to obtain a total count of all rows in a table, by simply executing:

```
SELECT COUNT(*) FROM dbo.Contacts;
```

If you specify a column name, as we did earlier, you *must* specify a `GROUP BY` clause. If you do not specify the `GROUP BY` clause, like in this statement:

```
SELECT AllowContactByPhone, COUNT(*) FROM dbo.Contacts;
```

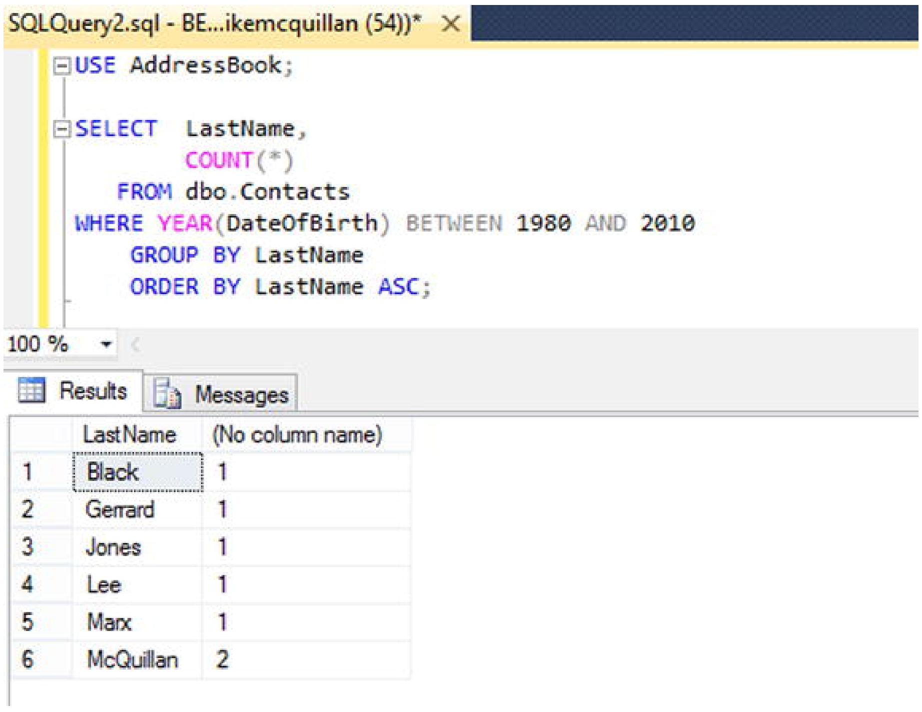
you'll see a message along these lines:

```
Msg 8120, Level 16, State 1, Line 3
Column 'dbo.Contacts.AllowContactByPhone' is invalid in the select list because it is not contained
```

`GROUP BY` can be used in conjunction with the `WHERE` and `ORDER BY` statements. It actually goes above the `ORDER BY`. Here's an example, bringing together all of our elements so far:

```
SELECT LastName, COUNT(*) FROM dbo.Contacts WHERE YEAR(DateOfBirth) BETWEEN 1980 AND 2010 GROUP BY
```

We are asking for the number of contacts born between 1980 and 2010 to be returned, broken down by their last name and also ordered by their last name. There are two McQuillans in this period, so as their names match it returns a count of 2 for that particular row (look at row 6 in Figure 11-11).


SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  LastName,
        COUNT(*)
    FROM dbo.Contacts
WHERE YEAR(DateOfBirth) BETWEEN 1980 AND 2010
     GROUP BY LastName
     ORDER BY LastName ASC;
```

|  | LastName | (No column name) |
|---|---|---|
| 1 | Black | 1 |
| 2 | Gerrard | 1 |
| 3 | Jones | 1 |
| 4 | Lee | 1 |
| 5 | Marx | 1 |
| 6 | McQuillan | 2 |

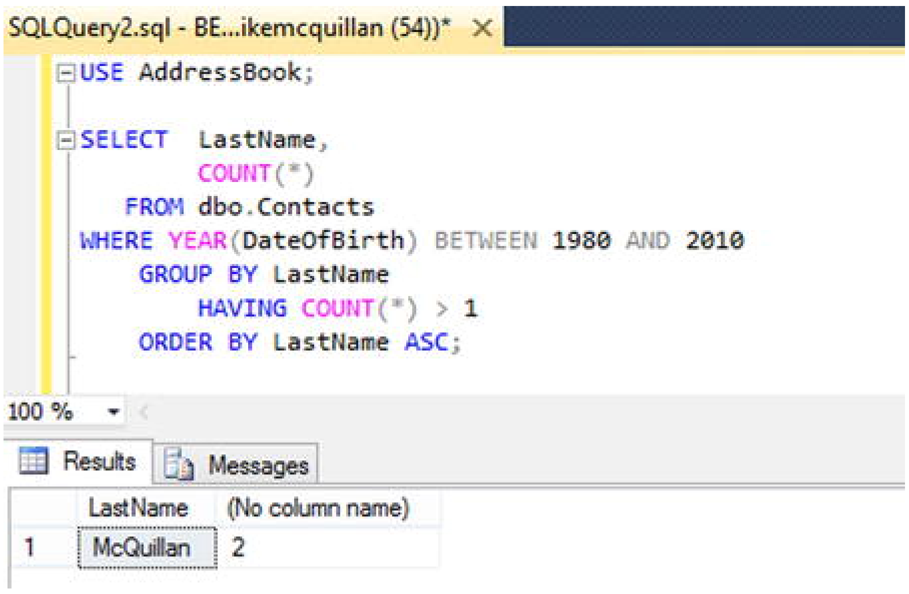
***Figure 11-11.*** *GROUP BY using WHERE and ORDER BY*

The GROUP BY clause has a few optional extras, the most commonly used of which is the HAVING clause. This acts as a WHERE clause for the GROUP BY clause. You can use it to limit results based on the output of the GROUP BY clause (you can also filter the results using ungrouped columns).

Modify our statement so it only returns results that had a count higher than 1:

```
SELECT LastName, COUNT(*) FROM dbo.Contacts WHERE YEAR(DateOfBirth) BETWEEN 1980 AND 2010 GROUP BY
```

Voila, only the McQuillan record now meets our criteria (see Figure 11-12). AND, OR, and other operators can be used in the HAVING clause, just like in the WHERE clause. It's powerful stuff!



SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  LastName,
        COUNT(*)
    FROM dbo.Contacts
WHERE YEAR(DateOfBirth) BETWEEN 1980 AND 2010
     GROUP BY LastName
        HAVING COUNT(*) > 1
     ORDER BY LastName ASC;
```

```
   LastName   (No column name)
1  McQuillan  2
```

*Figure 11-12.* `GROUP BY` *with* `HAVING` *clause*

We'll revisit the `GROUP BY` clause shortly when we look at `JOIN`s.

**Column Aliases**

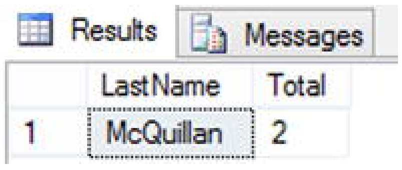
There was a problem in our various `GROUP BY` statements—did you spot it? The `COUNT(*)` column was displayed with the title `(No column name)`. That's not helpful—how will anybody know what the column is for? Fortunately, column aliases, which we met briefly in Chapter 10, can come to our rescue. To specify an alias, you use the `AS` keyword, followed by a name:

```
SELECT LastName, COUNT(*) AS Total FROM dbo.Contacts WHERE YEAR(DateOfBirth) BETWEEN 1980 AND 2010
```

To be truthful, the `AS` is optional, but if it weren't there you'd have:

```
COUNT(*) Total;
```

I find this confusing, so I always like to specify the `AS` so what I'm doing is obvious. It is also possible to alias tables, which we'll see in Chapter 12. Run this and the `COUNT(*)` column will have a name, as you can see in Figure 11-13. Cool!



```
   LastName   Total
1  McQuillan  2
```

*Figure 11-13. Giving a name to a calculated column*

**Limiting Rows with TOP**

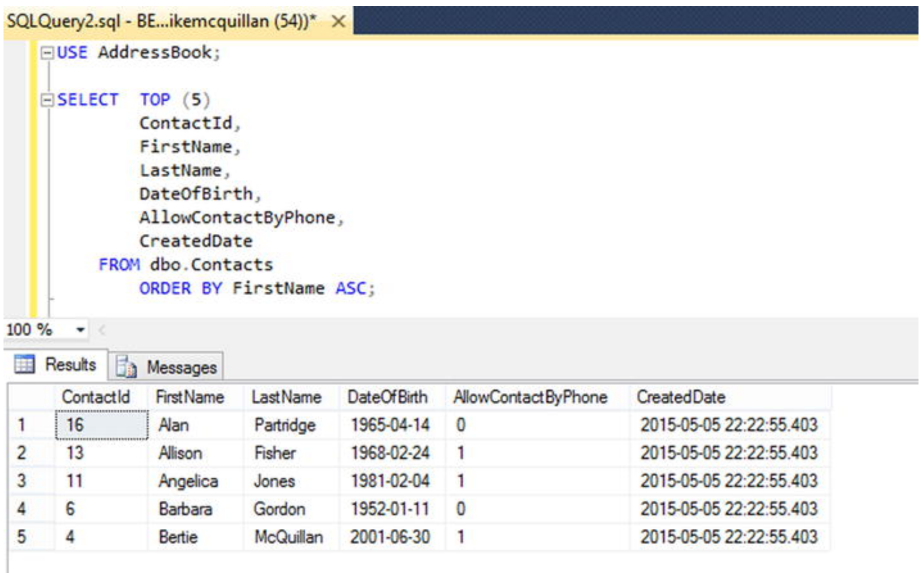
We've seen how we can filter the rows returned using the `WHERE` and `GROUP BY` clauses. We've also seen how the result sets can be ordered with `ORDER BY`. What do you do if you want to specify a maximum number of results? Say we want to grab the first five results from the `Contacts` table? No problem, we can specify a `TOP` clause.

The `TOP` clause does nothing more than say "Give me however many rows I ask for." If I say `TOP (10)`, I wil
receive 10 rows back. If the result set contains less than 10 rows, I'll receive however many were returned, bu
will never, ever receive more than 10.

Happily, `TOP` is as easy to implement as column aliases. This very simple query gives us the first five contacts in first name order:

```
SELECT TOP (5) ContactId, FirstName, LastName, DateOfBirth, AllowContactByPhone, CreatedDate FROM d
```

Execute this and just five rows are returned, which you can see in Figure 11-14.



SQLQuery2.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT  TOP (5)
        ContactId,
        FirstName,
        LastName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts
        ORDER BY FirstName ASC;
```

|  | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 16 | Alan | Partridge | 1965-04-14 | 0 | 2015-05-05 22:22:55.403 |
| 2 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 3 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 4 | 6 | Barbara | Gordon | 1952-01-11 | 0 | 2015-05-05 22:22:55.403 |
| 5 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |

***Figure 11-14.*** *Limiting result sets using* `TOP`

Note that the parentheses in `TOP (5)` are optional—we could have written `TOP 5` and the `SELECT` would have worked just as well. You can use `TOP` with `INSERT`, `UPDATE`, and `DELETE`, but for these you *must provide the parentheses*. It's also ANSI compliant, so it makes sense to be consistent.

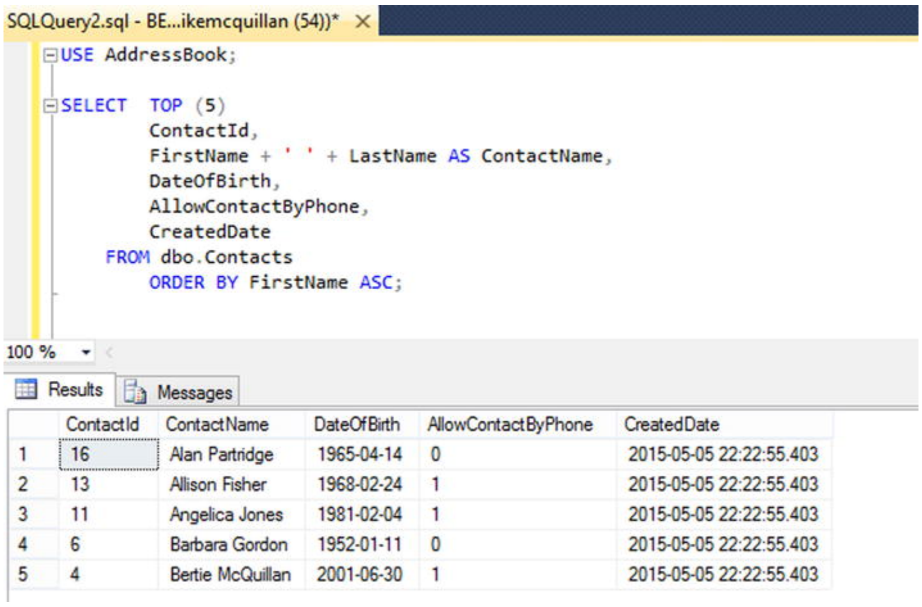
As a final `TOP` test, try changing `ORDER BY FirstName ASC;` to `ORDER BY FirstName DESC;`. The query will still return five rows, but you'll see five different rows because you have switched the order. `TOP` will only ever return the number of rows you specify, in the order you specify.

**Merging Columns**

Another nice trick you can use to make your `SELECT` statements stand out from the crowd is to combine multiple columns into one. This is pleasantly straightforward. What if we wanted to display a `ContactName` column instead of separate `FirstName` and `LastName` columns? Easy, we just concatenate the columns:

```
SELECT TOP (5) ContactId, FirstName + ' ' + LastName AS ContactName, DateOfBirth, AllowContactByPho
```

We are using the + sign to concatenate the value in the `FirstName` column with a space followed by the value in the `LastName` column. We use the `AS` keyword to specify a column alias, giving the column a name. Apart from the column alias this is very similar to what we saw when using the `PRINT` statement in Chapter 10. Run this and you'll have a `ContactName` column, as shown in Figure 11-15.



```
USE AddressBook;

SELECT  TOP (5)
        ContactId,
        FirstName + ' ' + LastName AS ContactName,
        DateOfBirth,
        AllowContactByPhone,
        CreatedDate
    FROM dbo.Contacts
        ORDER BY FirstName ASC;
```

|   | ContactId | ContactName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|
| 1 | 16 | Alan Partridge | 1965-04-14 | 0 | 2015-05-05 22:22:55.403 |
| 2 | 13 | Allison Fisher | 1968-02-24 | 1 | 2015-05-05 22:22:55.403 |
| 3 | 11 | Angelica Jones | 1981-02-04 | 1 | 2015-05-05 22:22:55.403 |
| 4 | 6 | Barbara Gordon | 1952-01-11 | 0 | 2015-05-05 22:22:55.403 |
| 5 | 4 | Bertie McQuillan | 2001-06-30 | 1 | 2015-05-05 22:22:55.403 |

***Figure 11-15.*** *Merging multiple columns together*

Columns you create in this manner are known as *derived columns* or *computed columns*. We'll see some more examples during our discussion of `JOIN`s in Chapter 12.

**Summary**

We've begun our travels in the world of `SELECT` statements. We haven't done anything particularly complicated yet, but we have covered all of the basics. We're not done with `SELECT` just yet, though—there's plenty more to come, starting in the very next chapter.

PREV
Chapter 10 : Creating Data Impo…

NEXT
Chapter 12 : Joining Tables