CHAPTER 8

DML (or Inserts, Updates, and Deletes)

We've created a solid database structure, but just as a database is useless without tables, so a table is useless without data. We've been using the table editor to add data to tables, but it's time to introduce the power of the T-SQL language. We'll look at how we can add data to our tables, edit it, and how we can remove it. We'll even do a little bit of querying, which will take us nicely into the next chapter.

Let's add some records!

Reference Data vs. Real Data

Before we add anything to the database, it's important to understand the difference between reference data and what I call real data. The **AddressBook** database contains two reference tables:

- `Roles`
- `PhoneNumberTypes`

and six real data tables:

- `Contacts`
- `ContactAddresses`
- `ContactNotes`
- `ContactPhoneNumbers`
- `ContactRoles`
- `ContactVerificationDetails`

Reference data represents data that you need, but that doesn't actually form part of your main data set. The intention of the **AddressBook** database is to store contact information. It doesn't have the aim of storing types of role or phone numbers—the fact that we are storing this information is a by-product of storing contact data. We store these types so we can link them to contacts, allowing us to determine whether a phone number is a work or home number, for example. But we could still use our contact data without this information—it is just used as a reference to embellish our contact data.

The real data tables hold the data we actually built the system for—in this case, contact data. If we built a stock control system we'd have a set of real data tables related to stock and order information, and then some reference tables storing things like item type, supplier category, and so on.

Okay, now that we've cleared that up, we can take a look at inserting some reference data, in preparation for adding real data.

Inserting Data

To prepare ourselves, let's rebuild the database. Close any query windows you may have open, and open `c:\temp\sqlbasics\rollback\00 - Rollback.sql`. Switch into SQLCMD mode (**Query ➤ SQLCMD Mode**) and run the script. Next, open `c:\temp\sqlbasics\apply\00 - Apply.sql`, go back into SQLCMD mode, and run it to recreate the database.

Okay, it's clean slate time. We'll start by adding some phone number types. We want to add:

- Home
- Work
- Mobile
- Other

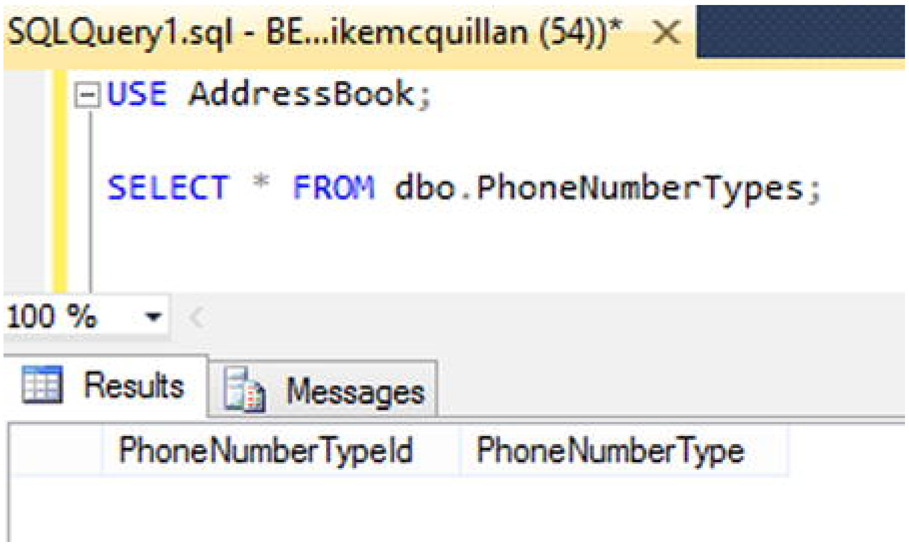
We have four phone number types to deal with. So far, we've been using the table editor to add records to tables. Now we are going to start using SQL DML statements.

WHAT IS DML?

DML stands for Data Manipulation Language. It represents a group of SQL statements that can be used to manage your data. There are four DML statements—`SELECT`, to retrieve data; `INSERT`, to add data; `UPDATE`, to update data (big surprise); and `DELETE`, to remove data (even bigger surprise). We'll look at `SELECT` in detail in the next chapter.

These statements are also known in the industry as CRUD—Create (`INSERT`), Read (`SELECT`), Update (`UPDATE`), Delete (`DELETE`). If somebody says CRUD to you, they're probably referring to the acronym, not the quality of your code!

Open up a New Query Window and type in and run the query shown in Figure 8-1.



```
USE AddressBook;

SELECT * FROM dbo.PhoneNumberTypes;
```

```
PhoneNumberTypeId    PhoneNumberType
```

Figure 8-1. *Querying an empty `PhoneNumberTypes` table*

As expected, no results are returned by the `SELECT` statement (if you did see some data, you need to recreate the database). We'll take a look at the `SELECT` statement in some detail in the next chapter, so for now just be aware that we use it to retrieve data from tables—specifically, here, the `PhoneNumberTypes` table.

There are three ways to use the `INSERT INTO` statement. The most common method of use is to use the `VALUES` keyword. In between the two statements in Figure 8-1, type this `INSERT INTO` statement:

```
INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');
```

`INSERT INTO` tells SQL Server what type of command we want to run—a record insert, in this case. `dbo.PhoneNumberTypes` is the name of the table we are inserting into, and we then add the names of the columns we want to insert into in parentheses. `PhoneNumberTypes` only has two columns, and the `PhoneNumberTypeId` is a calculated `IDENTITY` column for which we don't need to provide a value. So we just declare the `PhoneNumberType` column for insert. The next keyword is `VALUES`, which tells SQL Server we are now going to start providing the values for `INSERT`. These are also wrapped in parentheses, and we must provide the same number of columns and values. We have specified one column, so we must provide one value. If we tried to supply two values, we'd see:

```
Msg 110, Level 15, State 1, Line 3
```

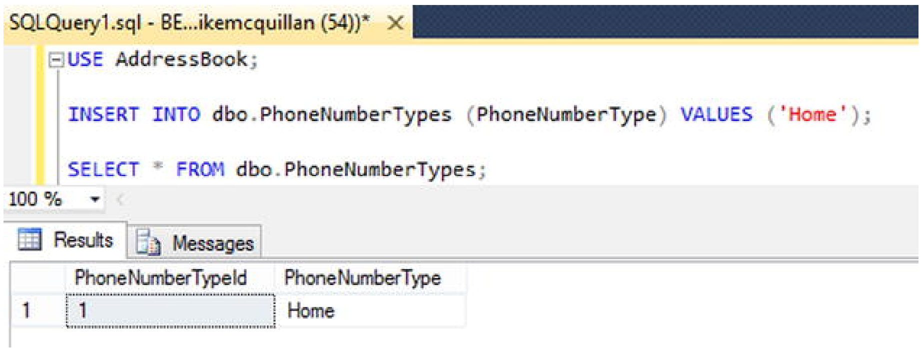
There are fewer columns in the `INSERT` statement than values specified in the `VALUES` clause. The number values in the `VALUES` clause must match the number of columns specified in the `INSERT` statement.

A simpler breakdown of the `INSERT INTO` statement we've just created is:

```
INSERT INTO TableName (Columns) VALUES (ValuesToInsert);
```

If there are multiple columns and values, you separate them with a comma. We'll see this shortly.

Now that we have added the `INSERT` statement to our script, run it. This time, the one record shown in Figure 8-2 should be returned.



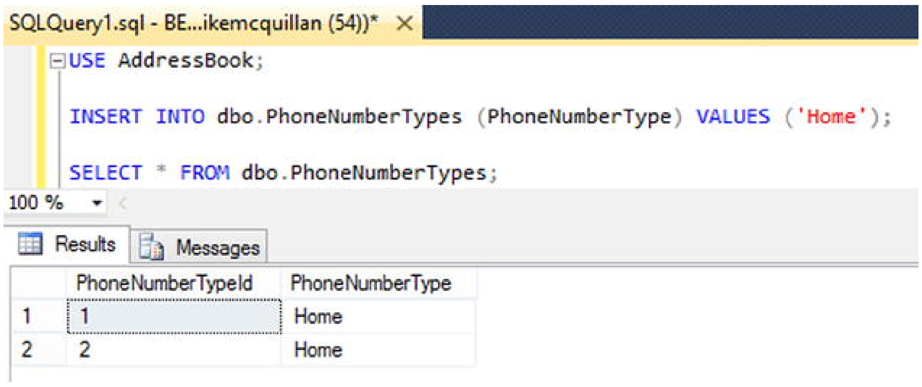
SQLQuery1.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');

SELECT * FROM dbo.PhoneNumberTypes;
```

	PhoneNumberTypeId	PhoneNumberType
1	1	Home

Figure 8-2. *Inserting a `Home` record into `PhoneNumberTypes`*

If we run this again, what do you think will happen? You may be surprised—a second record for `Home` will be inserted (Figure 8-3).



SQLQuery1.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');

SELECT * FROM dbo.PhoneNumberTypes;
```

	PhoneNumberTypeId	PhoneNumberType
1	1	Home
2	2	Home

Figure 8-3. *Inserting a duplicate `Home PhoneNumberType` record*

Eh? That wasn't in the script! Well actually, it was. There were a couple of ways we could have prevented this:

- Put a unique constraint on the PhoneNumberType column.
- Check if the value exists in the table before we try to insert it.

We've seen both of these techniques in the book already. We created a Unique index on the `ContactPhoneNumbers` table in the last chapter, and we've already used `IF` statements to check if particular databases and tables exist.

We're not going to add a Unique constraint to prevent this; instead, we'll add a check to determine if the record already exists. Before we do this, we need to remove the duplicate record. Run this statement in its own window:

```
DELETE dbo.PhoneNumberTypes WHERE PhoneNumberTypeId = 2;
```

This will remove the duplicate—you should see this message:

```
(1 row(s) affected)
```

Close that window and modify the `INSERT` script to check if the `Home` record already exists.

```
USE AddressBook;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Home')
BEGIN
INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');
END;

SELECT * FROM dbo.PhoneNumberTypes;
```

Run this as many times as you want—it won't insert a duplicate. The `IF NOT EXISTS` statement is something we saw first in Chapter 3. All it does is check if a "Home" record exists—if it does, the `INSERT INTO` statement will not be executed.

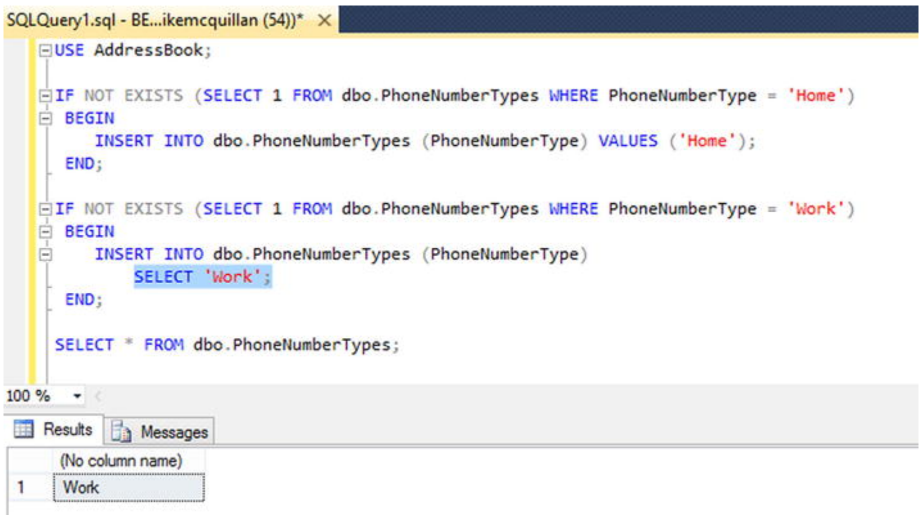
We'll add the second record—`Work`—using a slightly different mechanism. Enhance the script to include the `Work` insert:

```
USE AddressBook;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Home')
BEGIN
INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');
END;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Work')
BEGIN
INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType)
SELECT 'Work';
END;

SELECT * FROM dbo.PhoneNumberTypes;
```

We've added another `IF NOT EXISTS` check, this time for `Work`. But the `INSERT INTO` statement is slightly different. Instead of the `VALUES` keyword with brackets, this time we have `SELECT 'Work'`. Huh?

It's possible to insert multiple records in one go using the `SELECT` statement. We're going to investigate the `SELECT` statement in the very next chapter, but it can be used as part of the `INSERT` statement to insert one or more records (there's no real limit to the number of records you can insert using this). You can insert records from other tables, or specific values as we've done here.

If you highlight `SELECT 'Work'` in your Query Window and run it, one record will be returned, as shown in Figure 8-4:


SQLQuery1.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Home')
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');
 END;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Work')
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType)
        SELECT 'Work';
 END;

SELECT * FROM dbo.PhoneNumberTypes;
```

	(No column name)
1	Work

Figure 8-4. *Checking if records exist before inserting them*

It really is just a normal `SELECT` statement. We'll use this again soon.

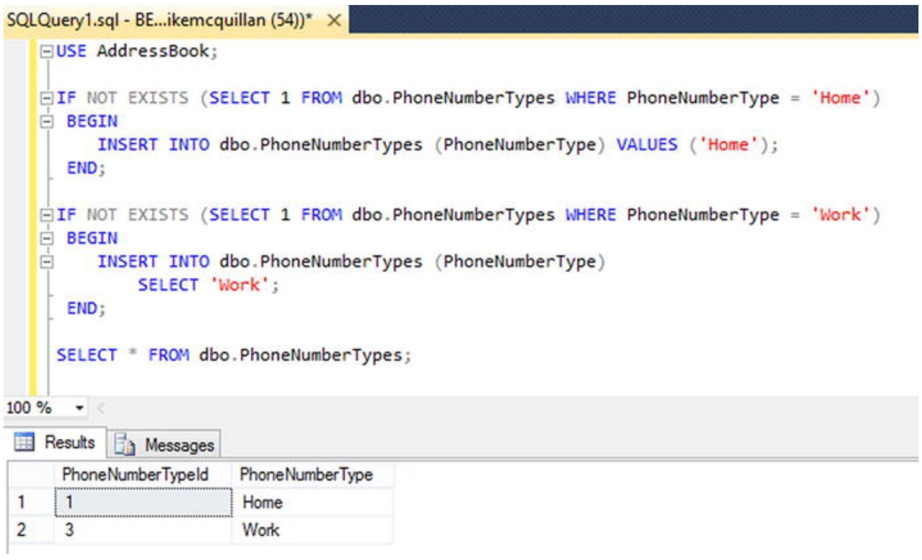
USING SELECT IN INSERT STATEMENTS

The preceding statement is using a literal value, `Work`. It is not inserting data from another table, as there is no `FROM` statement (we'll meet `FROM` in Chapter 11). When we use `SELECT` with a string enclosed in single quotes as we've done here, only the literal string is returned. Do not use double quotes, as this will return an error. You don't have to use literal strings—for instance, if you wanted to insert the current date/time into a column, you could use `SELECT GETDATE();` to insert it using the built-in SQL date/time function.

You can think of it as:

```
INSERT INTO TableName (Columns) SELECT ValuesToInsert
```

Click in your Query Window to unhighlight the code, and then run the entire script. The two records in Figure 8-5 should be returned.


SQLQuery1.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Home')
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');
 END;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Work')
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType)
        SELECT 'Work';
 END;

SELECT * FROM dbo.PhoneNumberTypes;
```

	PhoneNumberTypeId	PhoneNumberType
1	1	Home
2	3	Work

Figure 8-5. *Only inserting records when they don't already exist*

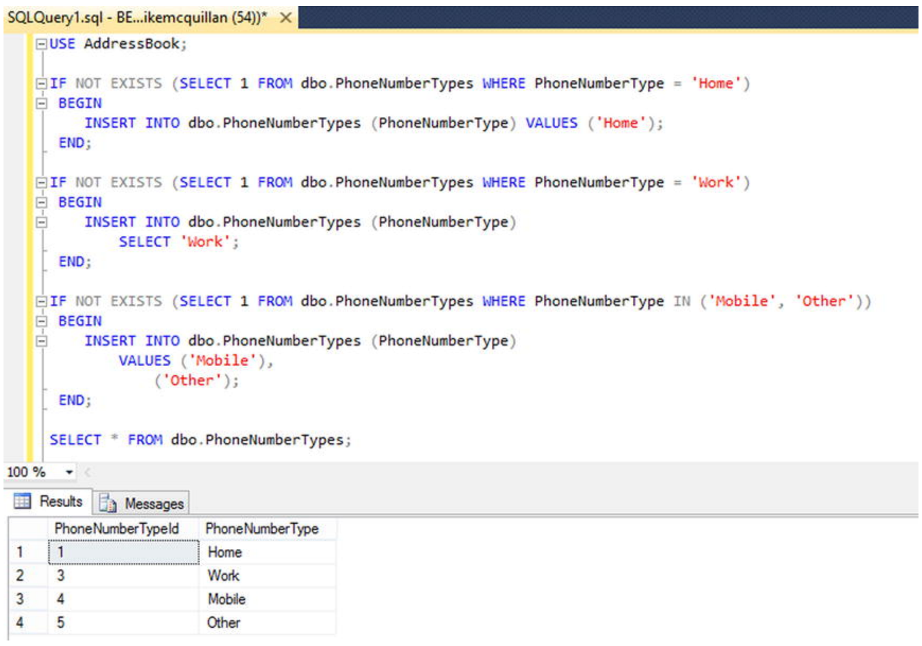
You may run this script as many times as you like, but only two records will ever return.

We have two more records to add. We'll use the third `INSERT INTO` option to do that. Type this above the `SELECT` statement:

```
IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType IN ('Mobile', 'Other'))
BEGIN
INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType)
VALUES ('Mobile'), ('Other');
END;
```

There are two things here we haven't seen. The first occurs in the `SELECT` statement used by `IF NOT EXISTS`. It uses `IN`, not `=`. The equals operator can be used to check for the existence of one value; the `IN` operator can check multiple values at the same time. We need to do this because the `INSERT INTO` statement is trying to insert two values, not one. You can provide multiple values by wrapping them up in their own set of parentheses, as you did earlier.

Run the entire script and you'll see four values have been inserted (Figure 8-6).



SQLQuery1.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Home')
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType) VALUES ('Home');
 END;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType = 'Work')
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType)
        SELECT 'Work';
 END;

IF NOT EXISTS (SELECT 1 FROM dbo.PhoneNumberTypes WHERE PhoneNumberType IN ('Mobile', 'Other'))
 BEGIN
    INSERT INTO dbo.PhoneNumberTypes (PhoneNumberType)
        VALUES ('Mobile'),
            ('Other');
 END;

SELECT * FROM dbo.PhoneNumberTypes;
```

	PhoneNumberTypeId	PhoneNumberType
1	1	Home
2	3	Work
3	4	Mobile
4	5	Other

Figure 8-6*. Inserting four* `PhoneNumberType` *records*

Again, you can keep running this—you'll only ever see four values returned.

We've now populated our `PhoneNumberTypes` table, and introduced the three different ways in which the `INSERT INTO` statement can be used. Before we create a script to insert records into the `Roles` table, finish this script off. Remove the `SELECT` statement and add a `GO` to the end of the script (we remove the `SELECT` as we don't need it when we rebuild the database). Save it as `c:\temp\sqlbasics\apply\10 - Insert PhoneNumberTypes.sql. Then open up the 00 - Apply.sql` script and add this code to the end of the script, so the records will be inserted if we rebuild the database again:

```
:setvar currentFile "10 - Insert PhoneNumberTypes.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

This should go above the line:

```
PRINT 'All apply scripts successfully executed.'
```

Wonderful! Now we'll create a new script to populate the `Roles` table.

More Inserts

Remember that roles are part of a many-to-many relationship with contacts, so one contact can hold one or more roles. We'll create a selection of roles that we can use in our system.

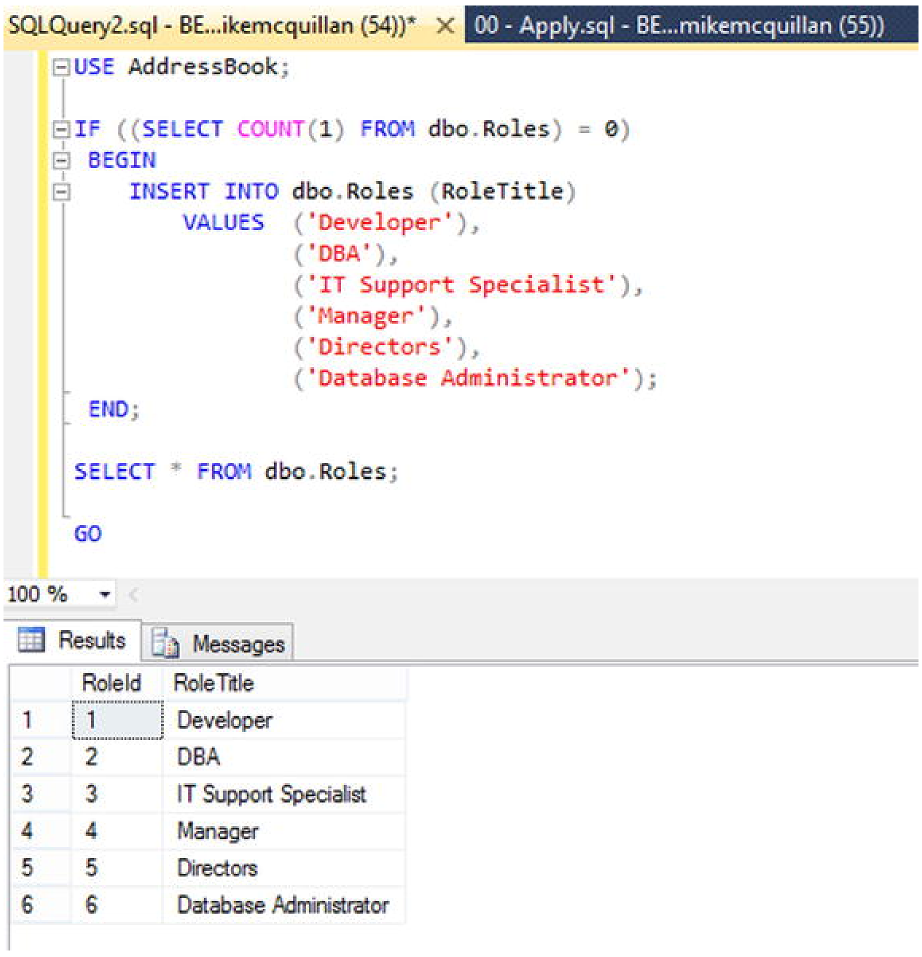
Open a New Query Window and add the role insert script.

```
USE AddressBook;

IF ((SELECT COUNT(1) FROM dbo.Roles) = 0)
BEGIN
INSERT INTO dbo.Roles (RoleTitle)
VALUES ('Developer'),
('DBA'),
('IT Support Specialist'),
('Manager'),
('Directors'),
('Database Administrator');
END;

SELECT * FROM dbo.Roles;
```

```
GO
```

We start with something new—there's no `IF NOT EXISTS` statement, just an `IF`. The `SELECT COUNT(1)` statement returns the number of rows in the `Roles` table. This line says if the number of rows found in the `Roles` table is zero, then insert the new rows. So if at least one row is present in `Roles` then nothing will be inserted. We're using the `VALUES` clause with multiple values to perform the inserts.

Run this and six rows should be inserted (Figure 8-7).



SQLQuery2.sql - BE...ikemcquillan (54))* × 00 - Apply.sql - BE...mikemcquillan (55))

```
USE AddressBook;

IF ((SELECT COUNT(1) FROM dbo.Roles) = 0)
 BEGIN
     INSERT INTO dbo.Roles (RoleTitle)
         VALUES  ('Developer'),
                 ('DBA'),
                 ('IT Support Specialist'),
                 ('Manager'),
                 ('Directors'),
                 ('Database Administrator');
 END;

SELECT * FROM dbo.Roles;

GO
```

	RoleId	RoleTitle
1	1	Developer
2	2	DBA
3	3	IT Support Specialist
4	4	Manager
5	5	Directors
6	6	Database Administrator

Figure 8-7*. An insert script for the* `Roles` *table*

Once you've seen this working, remove the `SELECT` from the script—again, we don't need it when rebuilding the database. Save the script as `c:\temp\sqlbasics\apply\11 - Insert Roles.sql`.

We're done with our reference data—or are we?

The UPDATE Statement

Oh no—we made a mistake when we inserted our records into the `Roles` table. Specifically, in record 5—it has the wrong `RoleTitle`. Nobody is going to have a RoleTitle of `Directors`—it should be `Director`. Darn! Well, our first fix is to open `c:\temp\sqlbasics\apply\11 - Insert Roles.sql` and remove the `s` from `Directors` (don't forget to save):

```
INSERT INTO dbo.Roles (RoleTitle)
VALUES ('Developer'),
('DBA')
('IT Support Specialist'),
('Manager'),
('Director'),
('Database Administrator');
```

The problem is, we've already executed this script. If we run it again, we still see `Directors` next to `RoleId 5`. This is because we added a check to the start of the script, preventing any inserts from occurring should at least one row exist. So how can we fix it? With an `UPDATE` statement! An `UPDATE` statement takes the form:

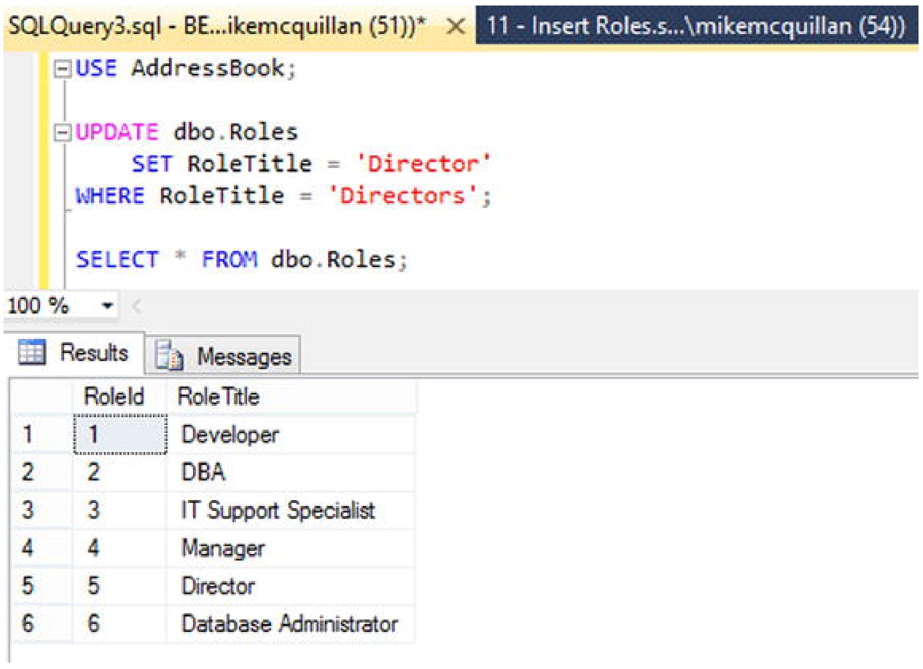
```
UPDATE TableName
SET ColumnName1 = NewValue1,
ColumnName2 = NewValue2,
ColumnNameN = NewValueN
WHERE RequiredColumnName = RequiredValue;
```

This is a pretty simple example, and the `WHERE` line is actually optional. You can specify as many columns for update as required. Here's our `UPDATE` script to fix `Directors`. Enter this into a New Query Window.

```
USE AddressBook;

UPDATE dbo.Roles
SET RoleTitle = 'Director'
WHERE RoleTitle = 'Directors';

SELECT * FROM dbo.Roles;
```

The first line is `UPDATE dbo.Roles`. This tells SQL Server the name of the table we want to update. The `SET` line is more granular, and informs the DBMS of the names of the columns we are changing, along with the new values. So we are setting the `RoleTitle` column to the value of `Director`.

The last line, the `WHERE` line, is key. The statement would run without this but would update the `RoleTitle` to `Director` *for every row in the table*. The `WHERE` clause limits the rows SQL Server will update. So here, we've told SQL Server to update rows where the `RoleTitle` column's value is equal to `Directors`. Run this and watch what happens—hopefully something similar to Figure 8-8.


SQLQuery3.sql - BE...ikemcquillan (51))*
11 - Insert Roles.s...\mikemcquillan (54))
100 %
Results
Messages

```
USE AddressBook;

UPDATE dbo.Roles
    SET RoleTitle = 'Director'
WHERE RoleTitle = 'Directors';

SELECT * FROM dbo.Roles;
```

	RoleId	RoleTitle
1	1	Developer
2	2	DBA
3	3	IT Support Specialist
4	4	Manager
5	5	Director
6	6	Database Administrator

Figure 8-8. *Updating the* `Directors` *role*

All fixed! You can run this again and again and the results won't change, as no rows now match the criteria we specified (`RoleTitle = 'Directors'`). Very nifty!

There are more complex versions of the `UPDATE` statement, but this example covers the basics. We'll meet those more complex versions in later chapters. The key thing to remember with `UPDATE` statements is to ensure your `WHERE` clauses are correct. Our later chapters will also show you how to check this.

As a final task here, modify the `00 - Apply.sql` script to include script 11. Here's the code you need to add above the final `PRINT` statement:

```
:setvar currentFile "11 - Insert Roles.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Deleting Records

We've added records and updated them. Logically, there's only one more thing we can do: remove them. Sometimes you do need to delete data from a database, perhaps for records that were mistakenly created. This is where the `DELETE` statement comes into play. This works in a very similar manner to the `UPDATE` statement. The format of the `UPDATE` statement is:

```
UPDATE TableName
SET ColumnName1 = NewValue1,
ColumnName2 = NewValue2,
ColumnNameN = NewValueN
WHERE RequiredColumnName = RequiredValue;
```

The `DELETE` statement is structured as:

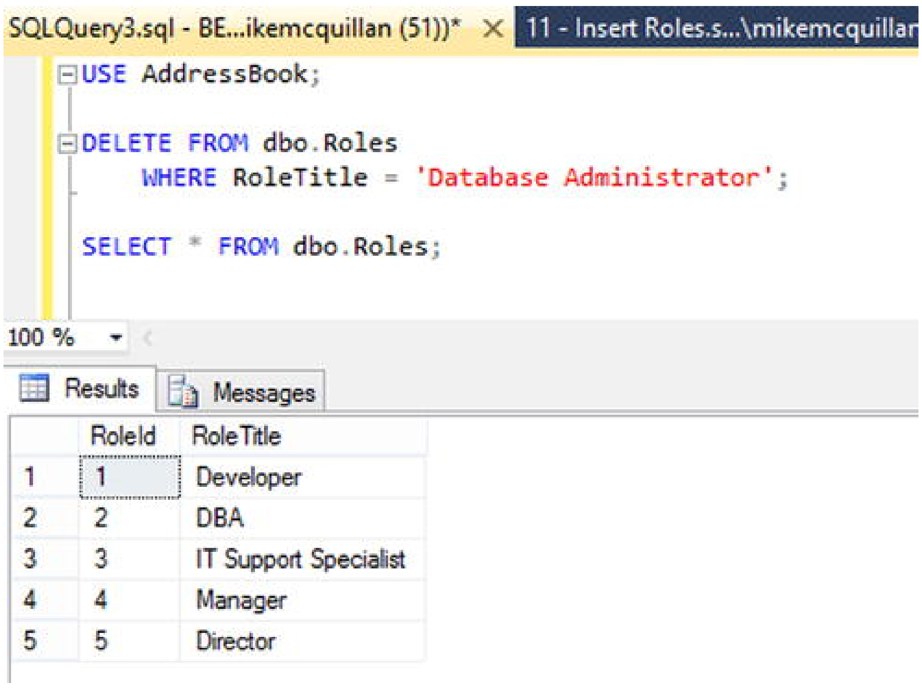
```
DELETE FROM TableName
WHERE RequiredColumnName = RequiredValue;
```

We don't have the `SET` section, as you can't delete individual column values—you delete the entire row.

Now, if you look back at the image of our `Roles` table, you'll see an error still remains. We've fixed the `Directors` problem, but we have duplicate items. Item 2 is `DBA` and item 6 is `Database Administrator`. These are one and the same thing, and having two separate items will knock out any reports we try to run in the future for DBAs. The solution is to delete one of these items. "DBA" is the standard industry term, so we'll delete the last item—`Database Administrator`. In a New Query Window, type in the `DELETE` script:

```
USE AddressBook;

DELETE FROM dbo.Roles
WHERE RoleTitle = 'Database Administrator';

SELECT * FROM dbo.Roles;
```

I should point out that the `SELECT` statements we are running at the end of each batch are purely optional—we're using them to inspect that our expected value has been deleted. Run this statement and just the five rows in Figure 8-9 will be returned.


SQLQuery3.sql - BE...ikemcquillan (51))*
11 - Insert Roles.s...\mikemcquillan
100 %
Results
Messages

```
USE AddressBook;

DELETE FROM dbo.Roles
    WHERE RoleTitle = 'Database Administrator';

SELECT * FROM dbo.Roles;
```

	RoleId	RoleTitle
1	1	Developer
2	2	DBA
3	3	IT Support Specialist
4	4	Manager
5	5	Director

Figure 8-9. *Deleting the* `Database Administrator` *role*

We need to remove the invalid record from our main `INSERT` script. Open up `c:\temp\sqlbasics\apply\11 - Insert Roles.sql`. Delete the `Database Administrator` line, and replace the ending comma on the `Director` line with a semicolon. Here's the updated script:

```
USE AddressBook;

IF ((SELECT COUNT(1) FROM dbo.Roles) = 0)
BEGIN
INSERT INTO dbo.Roles (RoleTitle)
VALUES  ('Developer'),
('DBA'),
('IT Support Specialist'),
('Manager'),
('Director');
END;

GO
```

We're nearly done; we just need to create rollback scripts to clear out the `PhoneNumberTypes` and `Rules` tables. These are pretty easy, as all we have to do is empty each table (neither table contained any data before we put it in there with our preceding scripts). Create script `c:\temp\sqlbasics\rollback\10 - Insert PhoneNumberTypes Rollback.sql` first:

```
USE AddressBook;

DELETE FROM dbo.PhoneNumberTypes;

GO
```

Not much to that, is there? We don't need a `WHERE` clause, as we want to empty the entire table. Of course, you should always be careful when running `DELETE` statements like this. I've accidentally emptied out tables more times than I care to remember! Never in production, though (a manager of mine did that once—it wasn't good!).

Save this script as `c:\temp\sqlbasics\rollback\11 - Insert Roles Rollback.sql`.

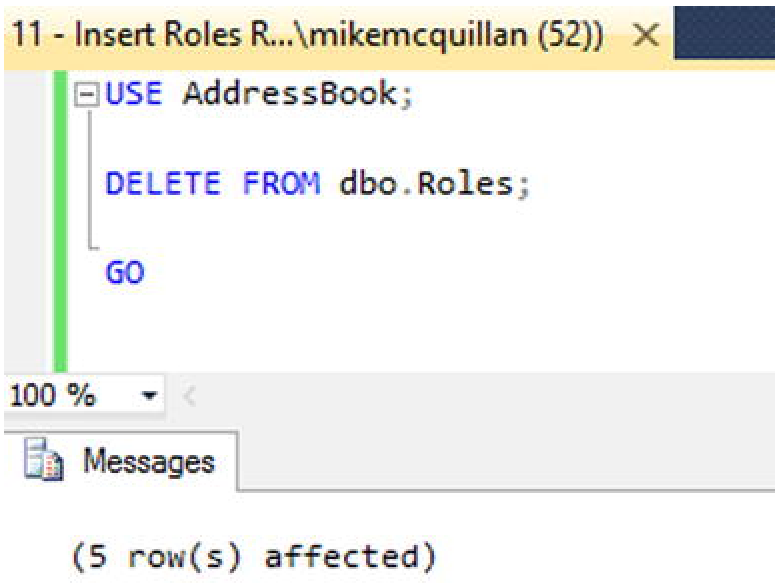
```
USE AddressBook;

DELETE FROM dbo.Roles;
```

```
GO
```

Another simple script. Finish off by adding both of these scripts to the top of the SQLCMD `00 - Rollback.sql` script, under the `:setvar` path line.

```
:setvar currentFile "11 - Insert Roles Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "10 - Insert PhoneNumberTypes Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Try running rollback scripts 10 and 11. You won't see any rows displayed, as we've not included a `SELECT` statement. You can tell the script has worked by the information message SQL Server returns to you. As Figure 8-10 shows, for instance, running the `Roles` rollback script (script 11) deletes five rows:



```
USE AddressBook;

DELETE FROM dbo.Roles;

GO
```

```
(5 row(s) affected)
```

Figure 8-10. *Deleting all records from the* `Roles` *table*

Magnificent!

Summary

This was an interesting chapter, as we moved from the creation of objects that store data to the creation of the data itself. We've covered how to insert, update, and delete data from tables, using a variety of options. We've also seen how we can use the `SELECT` statement to check if data already exists in the table before we try to insert it.

We've now populated our reference data, so our next job is to add some contact information to the system. And the `BULK INSERT` command is going to help us do just that!

PREV
Chapter 7 : NULLs and Table Co…

NEXT
Chapter 9 : Bulk Inserting Data