CHAPTER 10

Creating Data Import Scripts

We've just seen, in some detail, how the `BULK INSERT` statement can be used to import files into database tables. Unfortunately, there are some limitations with `BULK INSERT`, which meant we had to remove one of our contact records from the appropriate files.

Fortunately, T-SQL can come to our rescue. In this chapter, we are going to create a script to insert a contact record and its associated child records (addresses, phone numbers, and so on). I'll introduce lots of new T-SQL terms in this chapter, so hold your nose and jump in!

Purpose of the Import Script

Our aim is to successfully add Roald Dahl to the database, along with all his related records. The records we want to add are:

- `Contacts` record
- `ContactVerificationDetails` record
- `ContactRoles` record, adding Roald as a developer
- `ContactAddresses` record
- `ContactNotes` record
- Two `ContactPhoneNumbers` records, one for a mobile phone, and one for a work phone number

Roald failed to import as part of our `BULK INSERT` statement, as the `LastName` column had incorrectly been left blank. We could have created a second comma-separated file containing a fixed version of Roald Dahl's record, but then we wouldn't have fun creating this script, would we?

Starting the Script with Variables

In SSMS, open a New Query Window and add this T-SQL code:

```
USE AddressBook;

DECLARE @ContactId INT;
```

The `USE` statement is familiar to us, but the `DECLARE` statement is new. This declares a variable. A *variable* is a temporary object that can hold a value for us. The line breaks down as follows:

- **`DECLARE`**: Tells SQL Server we are declaring a variable.
- **`@ContactId`**: The name of the variable. Almost every variable in T-SQL must begin with an `@` symbol (there is the odd exception, but these exceptions are outside the scope of this book). A variable cannot be a reserved word (e.g., a word in the T-SQL language).
- **`INT`**: The type of variable. This can be a valid SQL Server data type, such as `VARCHAR(200)` or `DATETIME`.

Variables are very useful as they can give seemingly random values a meaningful name and purpose. They only exist for the lifetime of the script (at which point they are destroyed), and they can hold transitory information until it is needed. Variables will become an important part of your T-SQL toolkit. We'll expand the `DECLARE` statement to include a few other variables.

```
DECLARE @ContactId INT,
@DeveloperRoleId INT,
@MobilePhoneNumberTypeId TINYINT,
@WorkPhoneNumberTypeId TINYINT,
@NumberOfVerificationRecords INT,
```

```
@NumberOfContactRoleRecords INT,
@NumberOfContactAddressRecords INT,
@NumberOfContactNoteRecords INT,
@NumberOfContactPhoneNumberRecords INT;
```

Now we have nine variables. Nearly all of these are `INTs`, except for the two phone number type variables, which are `TINYINTs`. You should be able to figure out what each variable is for from its name. We could have written:

```
DECLARE @ContactId INT;
DECLARE @DeveloperRoleId INT;
DECLARE @MobilePhoneNumberTypeId TINYINT;
```

This is a lot more typing and not as succinct. It won't change the performance either way. We have a set of variables in place. Now we have to look up the reference data values we are interested in.

Reference Data Lookups

Do you remember our discussion about "real" data versus reference data in Chapter 8? Reference data was the stuff you add to the database that is used as metadata, such as phone number types and roles. We're not interested in these particular things themselves, but we are interested in them as part of a wider contact record.

We aim to add Roald Dahl as a developer, with mobile and work phone numbers. This necessitates looking up the appropriate reference data values. We need the ID values of the preceding records to be able to insert them into the appropriate tables. We are going to populate three of our variables with these values:

- `@DeveloperRoleId`
- `@MobilePhoneNumberTypeId`
- `@WorkPhoneNumberTypeId`

Add these lines under the variable declarations:

```
-- Obtain lookup values
SELECT @DeveloperRoleId = RoleId FROM dbo.Roles WHERE RoleTitle = 'Developer';
SELECT @MobilePhoneNumberTypeId = PhoneNumberTypeId FROM dbo.PhoneNumberTypes
WHERE PhoneNumberType = 'Mobile';
SELECT @WorkPhoneNumberTypeId = PhoneNumberTypeId FROM dbo.PhoneNumberTypes
WHERE PhoneNumberType = 'Work';
```

The first line begins with `--`. The text following it looks a bit odd—it reads like English. That's because it *is* English! This is what is known as a comment—something the developer adds to explain what a particular piece of code is doing. A comment is not part of the code, but instead helps to describe the code. We've helpfully stated here that the lines below the comment will obtain some lookup values. Comments are very useful and we'll sprinkle this script with them. Note there are two ways of declaring comments:

- Single-line comments can be created with `--` (two hyphens)
- Multi-line comments should be created using `/*` to start, and `*/` to finish

So you could have:

```
-- Obtain lookup values
/* Obtain lookup values */
```

Use whatever suits you best.

The second line is an almost typical `SELECT` statement:

```
SELECT @DeveloperRoleId = RoleId FROM dbo.Roles WHERE RoleTitle = 'Developer';
```

`SELECT @DeveloperRoleId = RoleId` says set the `@DeveloperRoleId` variable to the value of the `RoleId` column in the specified table.

`FROM dbo.Roles WHERE RoleTitle = 'Developer'` tells SQL Server to obtain `RoleId` from the Roles table, using the ID for the record where the role title is equal to `"Developer"`. This statement must only return one row, otherwise it will fail—you cannot assign multiple values to a variable.

We still haven't looked at `SELECT` in any great depth yet, but we did see it in basic use earlier, selecting out the contents of a table. Here we use it to assign a value to a variable. This is a pretty common piece of code and one you'll come across often. We could have written:

```
SELECT @DeveloperRoleId = 1;
```

This has same end result, but how do you know the ID of the developer role is `1`? And what happens if the developer role was added using a different ID? The first version is better—it tells us what it is trying to do and will always find the correct ID, thanks to the `WHERE` clause.

The two other `SELECT` statements are exactly the same, but populate the two phone number type variables.

Now add these lines.

```
PRINT 'Developer Role ID: ' + CAST(@DeveloperRoleId AS VARCHAR(20));
PRINT 'Mobile Phone Number Type ID: ' + CAST(@MobilePhoneNumberTypeId AS VARCHAR(20));
PRINT 'Work Phone Number Type ID: ' + CAST(@WorkPhoneNumberTypeId AS VARCHAR(20));
```

The `PRINT` statement is something else we've already met, but there is some new stuff, too. It's all normal at first—`PRINT 'Developer Role ID: '`. But then we have a + sign. What's that for? It is telling the `PRINT` statement to add whatever comes after the + sign to the `'Developer Role ID:'` string we've already declared (this is how SQL Server handles string concatenation). What comes after the plus sign is:

```
CAST(@DeveloperRoleId AS VARCHAR(20));
```

The `PRINT` statement can only display strings. `@DeveloperRoleId` is an `INT`. So if we had written this:

```
PRINT 'Developer Role ID: ' + @DeveloperRoleId;
```

The statement would have failed with an error. We need to change the type temporarily from `INT` to `VARCHAR`. The terminology to do this is to `CAST` the value from one type to another. So we cast the integer variable to a string using the `CAST` function.

THE CAST FUNCTION

You can cast most types from one to another, as long as they make sense. If you have a string value of `"2015-01-02"` you could cast it to a `DATETIME`, using `CAST('2015-01-02' AS DATETIME)`. But you couldn't cast `"Hello"` as a `DATETIME`.

When casting, think about what you are trying to do and whether you need the cast.

Also, take a look at the `CONVERT` function. This can be used like `CAST`, but it gives you more control over dates.

The two other `PRINT` statements are very similar to the statement we've just worked through.

We can run what we have so far by pressing F5. If you see the following output, everything is working!

```
Developer Role ID: 1
Mobile Phone Number Type ID: 3
Work Phone Number Type ID: 2
```

Splendid, our reference data lookups are working. Now to add some real data.

Inserting the Contact Record

We don't want to insert the contact record and its associated subrecords—*unless it doesn't already exist*. We already know how to check for the existence of a record: we use the `IF NOT EXISTS` statement we've been using throughout the book. This time, we want to check if Roald Dahl already exists, so we write the following line at the bottom of our script:

```
IF NOT EXISTS (SELECT 1 FROM dbo.Contacts WHERE FirstName = 'Roald' AND LastName = 'Dahl')
BEGIN;
```

If a `Contacts` record exists with a `FirstName` of `'Roald'` and a `LastName` of `'Dahl'`, we won't execute the code under the `BEGIN` block. `BEGIN` and `END` enclose the code that will execute if the condition required to fulfil the `IF NOT EXISTS` statement has been met.

We'll now add the `INSERT` under the `BEGIN;` statement we just added.

```
-- No need to specify AllowContactByPhone or CreatedDate, as they have
-- default values
-- and Roald has said we cannot contact him by phone
INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth) VALUES ('Roald', 'Dahl', '1916-09-13');
```

Nothing new here, we're just adding a record to `Contacts`. The next line is new, though:

```
SELECT @ContactId = @@IDENTITY;
```

The `ContactId` column in the `Contacts` table is an `IDENTITY` column. To be able to insert records into `ContactAddresses` and other subtables, we need the `ContactId`. Specifically, we need to store the `ContactId` value assigned to Roald Dahl's record in the `@ContactId` variable. This is where `@@IDENTITY` comes in. This is a SQL Server global variable. It holds the last identity value generated by the database, no matter which table generated it.

If you insert a record into `ContactAddresses` and it is given `AddressId 6` by its `IDENTITY` column, `@@IDENTITY` will hold the value `6`. If you insert a record into `Roles` and the `RoleId` is assigned `97`, `@@IDENTITY` will hold the value `97`.

When we insert Roald Dahl into the `Contacts` table, the record will be given an ID value. This will be assigned to `@@IDENTITY`. We then store the value of `@@IDENTITY` in `@ContactId` for future use.

WHY DON'T WE JUST USE `@@IDENTITY` ALL THE TIME?

You might be asking why we bother assigning `@@IDENTITY` to `@ContactId`. After all, `@@IDENTITY` is holding the value we need anyway, right? Well, maybe.

The first reason we use `@ContactId` is because it makes more sense for a developer to see `@ContactId` than `@@IDENTITY` peppered throughout our script.

The second reason is much more important. As soon as we *or anybody else* adds a new record that generates an `IDENTITY` value, the contents of `@@IDENTITY` will change. Imagine if our script runs and inserts a contact. Around the same time, another script ran and inserted a role. If we were using `@@IDENTITY`, our script would try inserting use the `RoleId`, which would lead to some pretty unpleasant errors. Always assign the contents of global variables you want to use to local variables immediately—then you can be confident the values are not going to change.

You can find out more about global variables in Appendix C.

Award yourself a prize if you noticed we haven't added an `END` to this block of code. We don't want to do this just yet—there are more inserts to come.

Adding Subrecords

Now that we have a main record, we can add the subrecords. We'll only do this if a valid value was assigned to the `@ContactId` variable. Here's the code to add the subrecords. This should go directly under the `SELECT @ContactId` line.

```
IF (@ContactId IS NOT NULL)
BEGIN

-- Add verification details
INSERT INTO dbo.ContactVerificationDetails(ContactId, DrivingLicenseNumber, ContactVerified)
VALUES (@ContactId, '1031', 0);

SELECT @NumberOfVerificationRecords = @@ROWCOUNT;

-- Add developer role to contact
IF (@DeveloperRoleId IS NOT NULL)
BEGIN
INSERT INTO dbo.ContactRoles(ContactId, RoleId) VALUES (@ContactId, @DeveloperRoleId);

SELECT @NumberOfContactRoleRecords = @@ROWCOUNT;
END;

-- Add an address
INSERT INTO dbo.ContactAddresses(ContactId, HouseNumber, Street, City, Postcode)
VALUES (@ContactId, '200', 'Shaftsbury Avenue', 'Hastings', 'TN38 8EZ');

SELECT @NumberOfContactAddressRecords = @@ROWCOUNT;

-- Add a note
INSERT INTO dbo.ContactNotes(ContactId, Notes)
VALUES (@ContactId, 'Roald Dahl is a famous author. He is best known for books aimed at children, s

SELECT @NumberOfContactNoteRecords = @@ROWCOUNT;

-- Add phone numbers
IF (@MobilePhoneNumberTypeId IS NOT NULL AND @WorkPhoneNumberTypeId IS NOT NULL)
BEGIN
INSERT INTO dbo.ContactPhoneNumbers(ContactId, PhoneNumberTypeId, PhoneNumber)
VALUES (@ContactId, @MobilePhoneNumberTypeId, '07100 988 199'),
 (@ContactId, @WorkPhoneNumberTypeId, '01424 700 700');

SELECT @NumberOfContactPhoneNumberRecords = @@ROWCOUNT;
END;

END;
```

There looks to be quite a bit here, but it's all pretty similar. Five inserts are happening in this code block:

- Adding a record to `ContactVerificationDetails`
- Adding a record to `ContactRoles`, if `@DeveloperRoleId` is valid
- Adding a record to `ContactAddresses`
- Adding a record to `ContactNotes`
- Adding two records to `ContactPhoneNumbers`, if both `@MobilePhoneNumberTypeId` and `@WorkPhoneNumberTypeId` are valid

The first line is `IF (@ContactId IS NOT NULL)`. This does nothing more than check if a valid value was assigned to the `@ContactId` variable through `@@IDENTITY`. If a valid value is present, the code enters the `IF` block, which is all of the code wrapped between the `BEGIN` and `END`.

The first insert is to the `ContactVerificationDetails` table. This is a typical `INSERT`, with nothing notable about it other than we are using `@ContactId` to specify the `ContactId` value to insert. The next line is interesting, though:

```
SELECT @NumberOfVerificationRecords = @@ROWCOUNT;
```

We are using another global variable here (all global variables begin with `@@`). This time it's `@@ROWCOUNT`. Th holds the number of the rows that were processed by the last DML statement to execute. We've just inserted

one row into `ContactVerificationDetails`, so we would expect `@@ROWCOUNT` to hold a value of one. We assign this to the `@NumberOfVerificationRecords` variable, so we can check at the end what the script did.

`@@ROWCOUNT` is a handy little variable and can be used to check if you have inserted, updated, or deleted the correct number of rows. It is reset after each statement, so you must assign it to a variable in the statement immediately after the DML command you executed.

Our next block of code combines everything we've seen so far.

```
-- Add developer role to contact
IF (@DeveloperRoleId IS NOT NULL)
BEGIN
INSERT INTO dbo.ContactRoles(ContactId, RoleId) VALUES (@ContactId, @DeveloperRoleId);

SELECT @NumberOfContactRoleRecords = @@ROWCOUNT;
END;
```

This code will add a record into `ContactRoles`, which you will recall is a many-to-many table. The `INSERT` only executes if the `@DeveloperRoleId` is not null; that is, it contains a valid value. If the `INSERT` succeeds we again store the `@@ROWCOUNT` value, this time in a different variable.

The next two `INSERT`s add records to both `ContactAddresses` and `ContactNotes`.

```
-- Add an address
INSERT INTO dbo.ContactAddresses(ContactId, HouseNumber, Street, City, Postcode)
VALUES (@ContactId, '200', 'Shaftsbury Avenue', 'Hastings', 'TN38 8EZ');

SELECT @NumberOfContactAddressRecords = @@ROWCOUNT;

-- Add a note
INSERT INTO dbo.ContactNotes(ContactId, Notes)
VALUES (@ContactId, 'Roald Dahl is a famous author. He is best known for books aimed at children, s

SELECT @NumberOfContactNoteRecords = @@ROWCOUNT;
```

These execute in exactly the same manner as the `ContactVerificationDetails INSERT`. The final `INSERT`, for `ContactPhoneNumbers`, is a little different from the others: it adds two records.

```
-- Add phone numbers
IF (@MobilePhoneNumberTypeId IS NOT NULL AND @WorkPhoneNumberTypeId IS NOT NULL)
BEGIN
INSERT INTO dbo.ContactPhoneNumbers(ContactId, PhoneNumberTypeId, PhoneNumber)
VALUES (@ContactId, @MobilePhoneNumberTypeId, '07100 988 199'),
(@ContactId, @WorkPhoneNumberTypeId, '01424 700 700');

SELECT @NumberOfContactPhoneNumberRecords = @@ROWCOUNT;
END;
```

The `INSERT`s only occur if the reference lookups successfully populated the two phone-number-type variables.

With all the `INSERT`s completed, you just need to add an `END` to the very end of the script, to close the `IF NOT EXISTS` block.

Displaying the Outcome

We finish the script with a `SELECT` statement, to display the `ContactId` we created, along with the number of records we inserted into each table.

```
-- Return what we've done
SELECT @ContactId AS ContactId,
@NumberOfVerificationRecords AS NumberOfVerificationRecordsAdded,
@NumberOfContactRoleRecords AS NumberOfContactRoleRecords,
@NumberOfContactAddressRecords AS NumberOfContactAddressRecords,
```

```
@NumberOfContactNoteRecords AS NumberOfContactNoteRecords,
@NumberOfContactPhoneNumberRecords AS NumberOfContactPhoneNumberRecords;
```

This is a fairly simple `SELECT` statement, as it doesn't use a table to obtain its data—it just displays the values of variables. Interestingly, each line has an extra `AS` clause at the end of it (e.g. `@ContactId AS ContactId`). This is a *column alias*, and is the column name that will be displayed on screen. We'll learn more about aliases in Chapter 11.

To complete the script, add a `GO` at the end. Here's the complete script.

```
USE AddressBook;

DECLARE @ContactId INT,
@DeveloperRoleId INT,
@MobilePhoneNumberTypeId TINYINT,
@WorkPhoneNumberTypeId TINYINT,
@NumberOfVerificationRecords INT,
@NumberOfContactRoleRecords INT,
@NumberOfContactAddressRecords INT,
@NumberOfContactNoteRecords INT,
@NumberOfContactPhoneNumberRecords INT;

-- Obtain lookup values
SELECT @DeveloperRoleId = RoleId FROM dbo.Roles WHERE RoleTitle = 'Developer';
SELECT @MobilePhoneNumberTypeId = PhoneNumberTypeId FROM dbo.PhoneNumberTypes WHERE PhoneNumberType
SELECT @WorkPhoneNumberTypeId = PhoneNumberTypeId FROM dbo.PhoneNumberTypes WHERE PhoneNumberType =

PRINT 'Developer Role ID: ' + CAST(@DeveloperRoleId AS VARCHAR(20));
PRINT 'Mobile Phone Number Type ID: ' + CAST(@MobilePhoneNumberTypeId AS VARCHAR(20));
PRINT 'Work Phone Number Type ID: ' + CAST(@WorkPhoneNumberTypeId AS VARCHAR(20));

IF NOT EXISTS (SELECT 1 FROM dbo.Contacts WHERE FirstName = 'Roald' AND LastName = 'Dahl')
BEGIN

-- No need to specify AllowContactByPhone or CreatedDate,
-- as they have default values
-- and Roald has said we cannot contact him by phone
INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth) VALUES ('Roald', 'Dahl', '1916-09-13');

SELECT @ContactId = @@IDENTITY;

IF (@ContactId IS NOT NULL)
BEGIN

-- Add verification details
INSERT INTO dbo.ContactVerificationDetails(ContactId, DrivingLicenseNumber, ContactVerified) VALUES

SELECT @NumberOfVerificationRecords = @@ROWCOUNT;

-- Add developer role to contact
IF (@DeveloperRoleId IS NOT NULL)
BEGIN
INSERT INTO dbo.ContactRoles(ContactId, RoleId) VALUES (@ContactId, @DeveloperRoleId);

SELECT @NumberOfContactRoleRecords = @@ROWCOUNT;
END;

-- Add an address
INSERT INTO dbo.ContactAddresses(ContactId, HouseNumber, Street, City, Postcode)
VALUES (@ContactId, '200', 'Shaftsbury Avenue', 'Hastings', 'TN38 8EZ');

SELECT @NumberOfContactAddressRecords = @@ROWCOUNT;

-- Add a note
INSERT INTO dbo.ContactNotes(ContactId, Notes)
VALUES (@ContactId, 'Roald Dahl is a famous author. He is best known for books aimed at children, s

SELECT @NumberOfContactNoteRecords = @@ROWCOUNT;

-- Add phone numbers
IF (@MobilePhoneNumberTypeId IS NOT NULL AND @WorkPhoneNumberTypeId IS NOT NULL)
BEGIN
INSERT INTO dbo.ContactPhoneNumbers(ContactId, PhoneNumberTypeId, PhoneNumber)
VALUES (@ContactId, @MobilePhoneNumberTypeId, '07100 988 199'),
(@ContactId, @WorkPhoneNumberTypeId, '01424 700 700');
```

```
SELECT @NumberOfContactPhoneNumberRecords = @@ROWCOUNT;
END;

END;

END;

-- Return what we've done
SELECT @ContactId AS ContactId,
@NumberOfVerificationRecords AS NumberOfVerificationRecordsAdded,
@NumberOfContactRoleRecords AS NumberOfContactRoleRecords,
@NumberOfContactAddressRecords AS NumberOfContactAddressRecords,
@NumberOfContactNoteRecords AS NumberOfContactNoteRecords,
@NumberOfContactPhoneNumberRecords AS NumberOfContactPhoneNumberRecords;

GO
```

Run this and you'll see a table of results that hopefully match Figure 10-1.

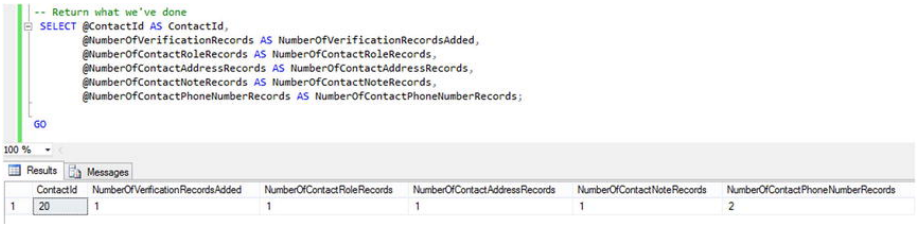

Figure 10-1. *Results of the import script*

Those numbers look good to me! Save this script as `c:\temp\sqlbasics\apply\18 - Insert Contact Record.sql`, and add this code to the end of the `00 - apply.sql` script:

```
:setvar currentFile "18 - Insert Contact Record.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Try running script 18 again. It will execute successfully, but you'll have a different set of results, which can be seen in Figure 10-2.

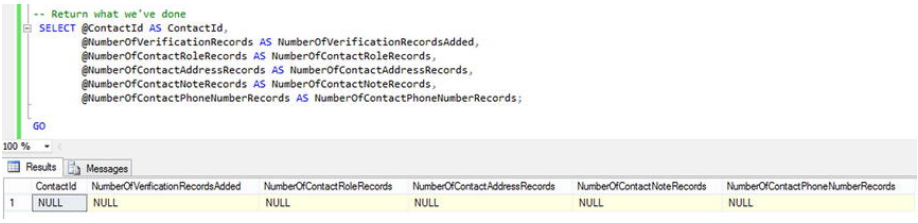

Figure 10-2. *Second set of results from the import script*

This is because we are checking if the record already exists. We've already executed the script once, so the second time it runs there is nothing for it to do. Hence the `NULL` values in every column.

As we are happy with the script, we no longer require the `SELECT` statement—it will just interfere with the output when we run our main apply script. Comment it out. Start by highlighting the code, then choose one of these SSMS options:

- Press Ctrl+K, then Ctrl+C (to uncomment, press Ctrl+K, then Ctrl+U)
- In the **Edit** menu, select **Advanced** ➤ **Comment Selection** (choose **Uncomment Selection** to uncomment the code)

Run the script again to prove that the `SELECT` statement no longer executes (Figure 10-3). If you ever need t put it back you can simply uncomment the code.

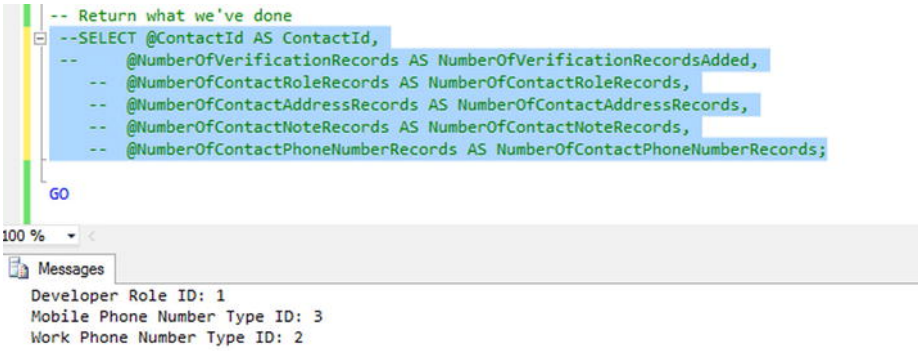

Figure 10-3. *Commented out code doesn't run*

Save your changes. Now we need to create a rollback script that will remove Roald Dahl.

The Rollback Script

The rollback script is much easier to write than the apply script. Here it is in all its glory.

```
USE AddressBook;

DECLARE @ContactId INT,
@NumberOfVerificationRecords INT,
@NumberOfContactRoleRecords INT,
@NumberOfContactAddressRecords INT,
@NumberOfContactNoteRecords INT,
@NumberOfContactPhoneNumberRecords NT;

IF EXISTS (SELECT 1 FROM dbo.Contacts WHERE FirstName = 'Roald' AND LastName = 'Dahl')
BEGIN

-- Obtain the contact ID value
SELECT @ContactId = ContactId FROM dbo.Contacts WHERE FirstName = 'Roald' AND LastName = 'Dahl';

-- Delete sub-records
DELETE FROM dbo.ContactVerificationDetails WHERE ContactId = @ContactId;
SELECT @NumberOfVerificationRecords = @@ROWCOUNT;

DELETE FROM dbo.ContactRoles WHERE ContactId = @ContactId;
SELECT @NumberOfContactRoleRecords = @@ROWCOUNT;

DELETE FROM dbo.ContactAddresses WHERE ContactId = @ContactId;
SELECT @NumberOfContactAddressRecords = @@ROWCOUNT;

DELETE FROM dbo.ContactNotes WHERE ContactId = @ContactId;
SELECT @NumberOfContactNoteRecords = @@ROWCOUNT;

DELETE FROM dbo.ContactPhoneNumbers WHERE ContactId = @ContactId;
SELECT @NumberOfContactPhoneNumberRecords = @@ROWCOUNT;

-- Delete main record
DELETE FROM dbo.Contacts WHERE ContactId = @ContactId;

END;

--SELECT @ContactId AS DeletedContactId,
-- @NumberOfVerificationRecords AS NumberOfVerificationRecordsAdded,
-- @NumberOfContactRoleRecords AS NumberOfContactRoleRecords,
-- @NumberOfContactAddressRecords AS NumberOfContactAddressRecords,
-- @NumberOfContactNoteRecords AS NumberOfContactNoteRecords,
-- @NumberOfContactPhoneNumberRecords AS NumberOfContactPhoneNumberRecords;

GO
```

We have a commented-out `SELECT` statement at the end. This is the same as our apply script—we've added the `SELECT` for any troubleshooting we might need to do at a later date.

Save this to `c:\temp\sqlbasics\rollback\18 - Insert Contact Record Rollback.sql`, and add the execution code to the top of the `00 - Rollback.sql` script.

```
:setvar currentFile "18 - Insert Contact Record Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

We've seen everything this script does before. The variable declarations are very similar to those we saw in the apply script; we just don't have as many of them as we don't need to look up reference data values.

The next line is the key to the entire script:

```
IF EXISTS (SELECT 1 FROM dbo.Contacts WHERE FirstName = 'Roald' AND LastName = 'Dahl')
```

This tells us the script will only do something if a record called Roald Dahl exists in the `Contacts` table. If it doesn't, the script will execute the `SELECT` statement after the `END` statement and a lot of `NULL` values will be displayed. If it does exist, the code enters the `IF` block and obtains the `ContactId` value for Roald Dahl, assigning it to the `@ContactId` variable.

```
-- Obtain the contact ID value
SELECT @ContactId = ContactId FROM dbo.Contacts WHERE FirstName = 'Roald' AND LastName = 'Dahl';
```

We then execute `DELETE` statements against each table, storing the number of rows processed by each statement. Note that we don't check if `@ContactId` has a valid value here. We could do that but it doesn't matter. If the `@ContactId` variable is NULL the `DELETE` statements will execute but they won't do anything, as no `ContactId` with a `NULL` value can possibly exist.

The final `DELETE` statement removes the main record from the `Contacts` table.

```
-- Delete main record
DELETE FROM dbo.Contacts WHERE ContactId = @ContactId;
```

And as with the apply script, we have a `SELECT` statement that shows us how many records were deleted.

Uncommend the `SELECT` statement at the bottom and run the script. We should see that `ContactId 20` is the `DeletedContactId`, along with numbers in the other columns that denote how many records we removed from each table. In other words, your results should look like those in Figure 10-4.

Figure 10-4. *Showing what was deleted*

Run it again, and just like the apply statement, a lot of `NULL` values will be returned (you can see these in Figure 10-5). This is because of the `IF EXISTS` check at the top of the script—the script will only attempt to delete if Roald Dahl exists. He doesn't, as we deleted him when we first ran the script.

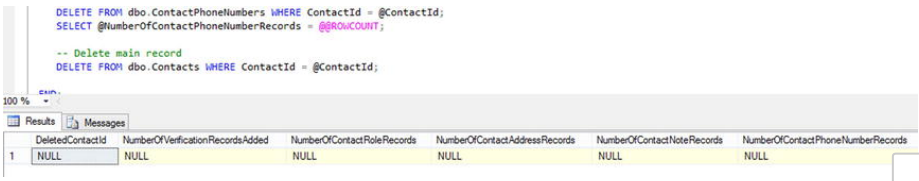

Figure 10-5. *Running the delete when it has already been executed*

Comment the `SELECT` statement back out, save the script, and we're all done! Rebuild the database so you are ready for the next chapter.

Summary

This chapter has concluded our look at data imports. Strictly speaking, what we did in this chapter wasn't a data import per se, but rather a look at how an individual record, along with its child records, can be added to a database. But we've now seen how to import using files and how to add data using scripts.

We used the `SELECT` statement a few times in this chapter, and indeed we've come across this statement before. It's now time to take a proper look at the `SELECT` statement. Let's see what powers it can grant us!

PREV
Chapter 9 : Bulk Inserting Data

NEXT
Chapter 11 : The SELECT Statem…