CHAPTER 15

Transactions

What do you think of when you hear the word “transaction”? I think of somebody who either buys or sells something to someone else. That’s not quite what a transaction is in SQL Server, but as we’ll see, this example will help us understand how transactions can be used to ensure our data stays consistent. Time to transact. . . .

What Is a Transaction?

I’ve just defined “transaction” as somebody who either buys something from or sells something to someone else. In this kind of encounter, two things occur:

- The buyer provides a payment of some kind
- The seller provides an item or service of some kind

If the buyer does not provide a payment, the seller does not provide the item or service. Likewise, if the seller does not provide the item or service, the buyer will not provide the payment. For the transaction to work, both elements—the buyer and seller—must complete their parts of the deal. Failure to do this will result in a failed transaction.

What SQL Server does with transactions is very similar. We specify a number of actions that must occur. If they all succeed, the transaction succeeds. Should one of them fail, the transaction fails and we can take the decision to roll back the changes, putting the data back to how it was before any changes were made. We’ll see how to put this together in SQL Server.

ACID

No chapter on transactions would be complete without mentioning ACID. This is not some kind of 1990s dance movement (look it up, kids); rather, it’s an acronym for the things that guarantee the reliable processing of database transactions. ACID stands for:

- Atomicity
- Consistency
- Isolation
- Durability

I know, it just rolls off the tongue. Atomicity specifies that either all of the transaction succeeds, or none of it succeeds. Consistency dictates that the transaction will keep the database in a valid state, and that any data provided by the transaction meets database rules, such as constraints.

Multiple transactions may run at the same time. This is where Isolation comes in, as it ensures all transactions run completely independently of each other. Finally, we have Durability. This property of ACID states that once the transaction has been committed, it will be present in the database forever, no matter what happens; even a software or computer crash.

Together, these four properties form a guard against badly implemented transactions, and SQL Server incorporates these properties into its transaction implementation.

A Transaction Example

We are going to use two tables in our example: the `Contacts` table, and the `ContactVerificationDetails` table. These exist together in a one-to-one relationship. The rules here are as follows:

- A record should not exist in `ContactVerificationDetails` until a record exists in `Contacts`
- Once a record has been inserted into the `Contacts` table, a corresponding record should also be inser into the `ContactVerificationDetails` table

- If both records cannot be created within the same transaction, neither record should be created
- It should not be possible to create a `Contact` record without a `ContactVerificationDetails` record

What we want to say to SQL Server is the following:

- Create a `Contact` record
- If the `Contact` record was created successfully, create a `ContactVerificationDetails` record
- If the `ContactVerificationDetails` record is created successfully, save both records to the database
- If the creation of the `ContactVerificationDetails` record fails, remove the `Contact` record and put the database back to how it was before we started

Sounds simple enough. Let's meet the T-SQL statements that will help us meet these goals.

T-SQL Transaction Statements

T-SQL provides a number of transaction commands, but we will concentrate on the three core commands. These are the statements most developers use on a day-to-day basis.

BEGIN TRANSACTION

You use `BEGIN TRANSACTION` to start a transaction. This tells SQL Server that every statement you subsequently run should be seen as part of the current transaction.

You can give the transaction an optional name, although this is seldom used (in my experience, anyway). Giving the transaction a name could be useful if you were nesting transactions; that is, embedding one transaction in another. This is often used when calling stored procedures from within a transaction, and that stored procedure contains a transaction itself. In a nested transaction, you should be aware that even if the inner transaction completes successfully but the outer transaction rolls back, the work performed by the inner transaction will also be rolled back.

You write `BEGIN TRANSACTION` as:

```
BEGIN TRANSACTION;
```

or:

```
BEGIN TRANSACTION TransactionName;
```

COMMIT TRANSACTION

This signals the end of a transaction. At this point, all data written to the database during the transaction is committed (saved). When you `BEGIN` a transaction, this causes something called a lock to be opened on the tables involved in the transaction. Normally, this lock prevents other users from accessing the table during the transaction, so you want to run the `COMMIT TRANSACTION` command as quickly as possible.

To execute a commit, write:

```
COMMIT TRANSACTION;
```

To commit a named transaction, you provide the name at the end of the statement:

```
COMMIT TRANSACTION TransactionName;
```

ROLLBACK TRANSACTION

A rollback should be used when something doesn't go to plan. It is normally combined with some error-checking code that determines if everything is okay to proceed. If things aren't okay, the rollback kicks in. Fo example:

```
IF previous record created successfully
INSERT new record;
COMMIT TRANSACTION;
ELSE
ROLLBACK TRANSACTION;
```

When you call `ROLLBACK TRANSACTION`, all records that have been inserted, updated, or deleted within the transaction to that point are either removed or restored to their original state. `ROLLBACK TRANSACTION` puts the database back to exactly how it was before the transaction started.

Calling `ROLLBACK TRANSACTION` is very similar to the other transaction statements:

```
ROLLBACK TRANSACTION;
```

Or if you specified a transaction name:

```
ROLLBACK TRANSACTION TransactionName;
```

When Should I Use Transactions?

Strictly speaking, you should always use transactions. To be clear: you *do* always use transactions—you just don't know it half the time. There are two types of transaction: *implicit*, and *explicit*. Take a look at this statement:

```
INSERT INTO dbo.Roles (RoleTitle) VALUES ('Helpdesk Operator');
```

This will execute as an implicit transaction. This means SQL Server `BEGIN`s and `COMMIT`s a transaction just for this statement. It does it in the background. Because the `BEGIN` and `COMMIT` apply to one statement, there is no way to roll this change back—you would need to delete the row. Whenever we've executed an `INSERT`, `UPDATE` or `DELETE` during this book, we've been using implicit transactions.

Explicit transactions are what we are concerned with in this very chapter. These are created when you tell SQL Server where you want the transaction to start and finish.

```
BEGIN TRANSACTION;

INSERT INTO dbo.Roles (RoleTitle) VALUES ('Helpdesk Operator');

COMMIT TRANSACTION;
```

Here, we have the ability to roll the change back if necessary by adding some code before the `COMMIT TRANSACTION`. It's also much clearer what is going on.

I recommend you always wrap your DML statements in a transaction. Quite apart from the clarity this brings to your code, it makes code modifications easier in the future should you need to modify further records as part of the transaction.

What If I Don't Use Transactions?

As we've just seen, there is no requirement for you to use transactions. If you don't use them, each DML statement will execute in its own implicit transaction . . . but you may leave issues in your code that could easily be rectified by using a transaction. We'll work through an example that shows what happens when the use of transactions goes bad!

Creating a Transaction

It's time to create a transaction. Open up a New Query Window and type the code below. DO NOT RUN THIS CODE YET!

```
USE AddressBook;

DECLARE @ContactId INT;
```

```
BEGIN TRANSACTION;

INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone) VALUES('Laura', 'Ro

SELECT @ContactId = SCOPE_IDENTITY();

PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));
```

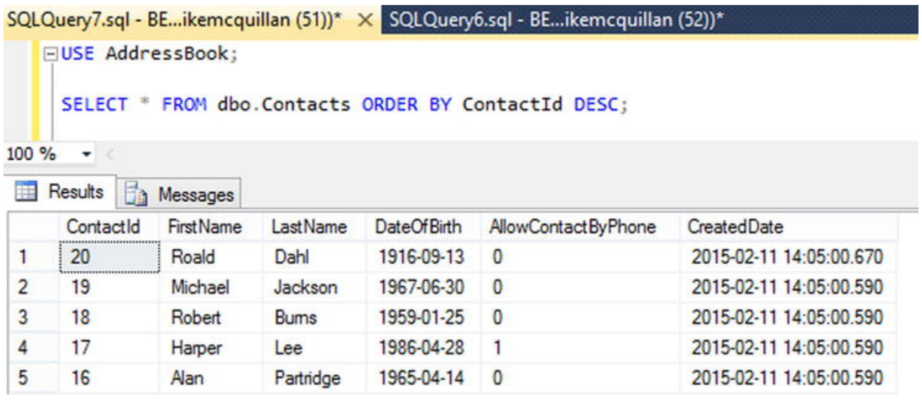
Apart from the `BEGIN TRANSACTION` statement, we've seen all this before. We are inserting a contact record and storing the ID generated for that contact in the `@ContactId` variable we declared.

Locking a Table with a Bad Transaction

Now, I shouted at you a moment ago not to run this script. Sorry about that, but I did have a good reason! This script presents us with a perfect opportunity to see what happens when transactions go wrong. Open a second query window and run this query:

```
USE AddressBook;

SELECT * FROM dbo.Contacts ORDER BY ContactId DESC;
```

It should return results successfully, with the most recent `ContactId` at the top. Just like the results in Figure 15-1, in fact.

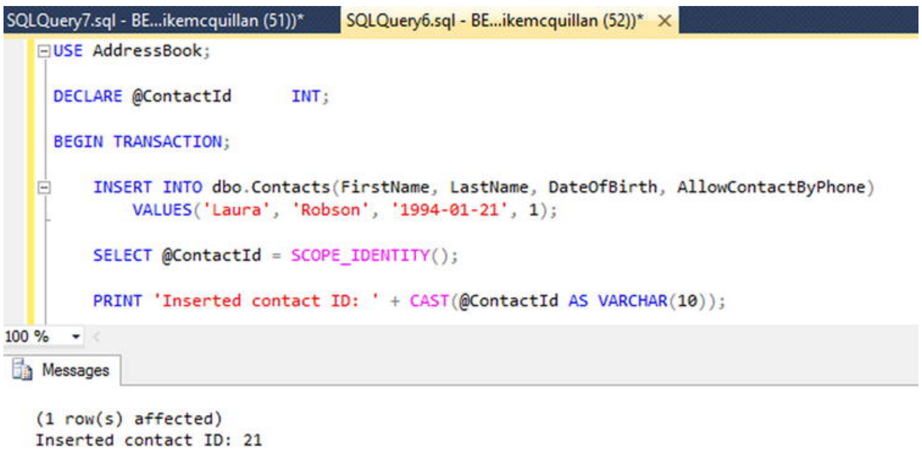

SQLQuery7.sql - BE...ikemcquillan (51))*
SQLQuery6.sql - BE...ikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

SELECT * FROM dbo.Contacts ORDER BY ContactId DESC;
```

	ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
1	20	Roald	Dahl	1916-09-13	0	2015-02-11 14:05:00.670
2	19	Michael	Jackson	1967-06-30	0	2015-02-11 14:05:00.590
3	18	Robert	Burns	1959-01-25	0	2015-02-11 14:05:00.590
4	17	Harper	Lee	1986-04-28	1	2015-02-11 14:05:00.590
5	16	Alan	Partridge	1965-04-14	0	2015-02-11 14:05:00.590

Figure 15-1. *Running a simple* `SELECT` *statement*

Now return to our transaction window and run the `BEGIN TRANSACTION` script. It should complete successfully, as demonstrated in Figure 15-2.


SQLQuery7.sql - BE...ikemcquillan (51))*
SQLQuery6.sql - BE...ikemcquillan (52))*
100 %
Messages

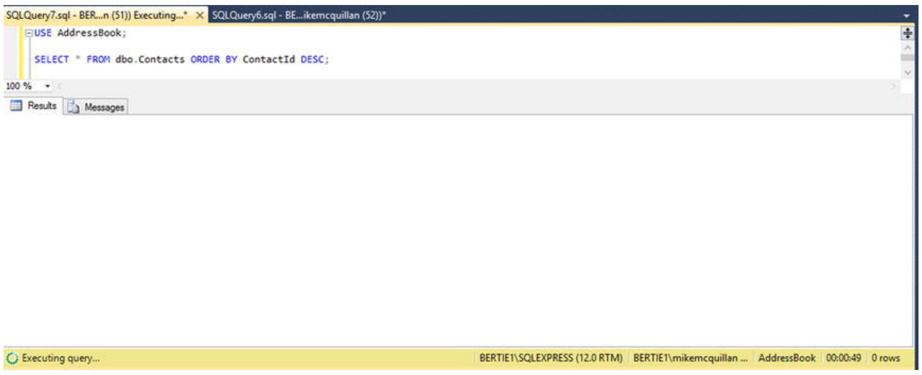
```
USE AddressBook;

DECLARE @ContactId      INT;

BEGIN TRANSACTION;

    INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone)
        VALUES('Laura', 'Robson', '1994-01-21', 1);

    SELECT @ContactId = SCOPE_IDENTITY();

    PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));
```

```
(1 row(s) affected)
Inserted contact ID: 21
```

Figure 15-2. *Successfully running a* `BEGIN TRANSACTION` *script*

The `PRINT` statement tells us that contact ID 21 has been successfully inserted. A great result—we inserted a contact and used a transaction to do it. All seems okay so far. Now switch back to the other window and run the `SELECT` statement there.

Uh-oh . . . nothing is being returned. All you can see is “Executing query . . .” in the bottom left-hand corner, as shown in Figure 15-3.


SQLQuery7.sql - BER...n (51)) Executing...*
SQLQuery6.sql - BE...ikemcquillan (52))*

```
USE AddressBook;

SELECT * FROM dbo.Contacts ORDER BY ContactId DESC;
```

```
Executing query...    BERTIE1\SQLEXPRESS (12.0 RTM) | BERTIE1\mikemcquillan ... | AddressBook | 00:00:49 | 0 rows
```

Figure 15-3. *The never-ending query*

In this example, the query has been running for 49 seconds so far! What is happening? Everything works fine in the other window.

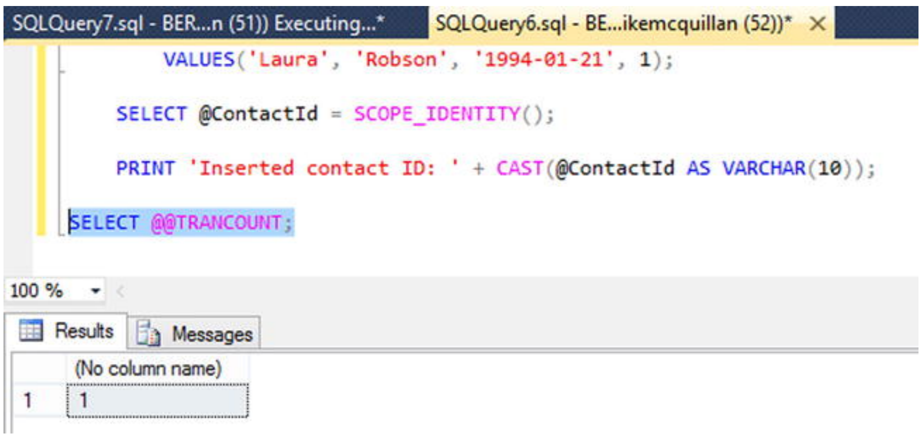
The window in which we wrote `BEGIN TRANSACTION` has a lock on the table. Locks are used to protect the integrity of your data and to maintain the ACID properties we were discussing earlier. We’ve begun the transaction but we haven’t committed it or rolled it back. As a result, the lock has not been released on the table.

@@TRANCOUNT

Return to the `BEGIN TRANSACTION` window. Paste this line of code at the bottom of the script, after the `PRINT` statement (don’t run it yet):

```
SELECT @@TRANCOUNT;
```

`@@TRANCOUNT` is another of those global system variables we’ve been meeting during our travels. This one tells you how many transactions are open for the current window. Highlight the `SELECT` statement you just pasted and run it. Sure enough, there is one transaction open. You can see this in Figure 15-4.


SQLQuery7.sql - BER...n (51)) Executing...*
SQLQuery6.sql - BE...ikemcquillan (52))*

```
        VALUES('Laura', 'Robson', '1994-01-21', 1);

    SELECT @ContactId = SCOPE_IDENTITY();

    PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));

SELECT @@TRANCOUNT;
```

```
   (No column name)
1  1
```

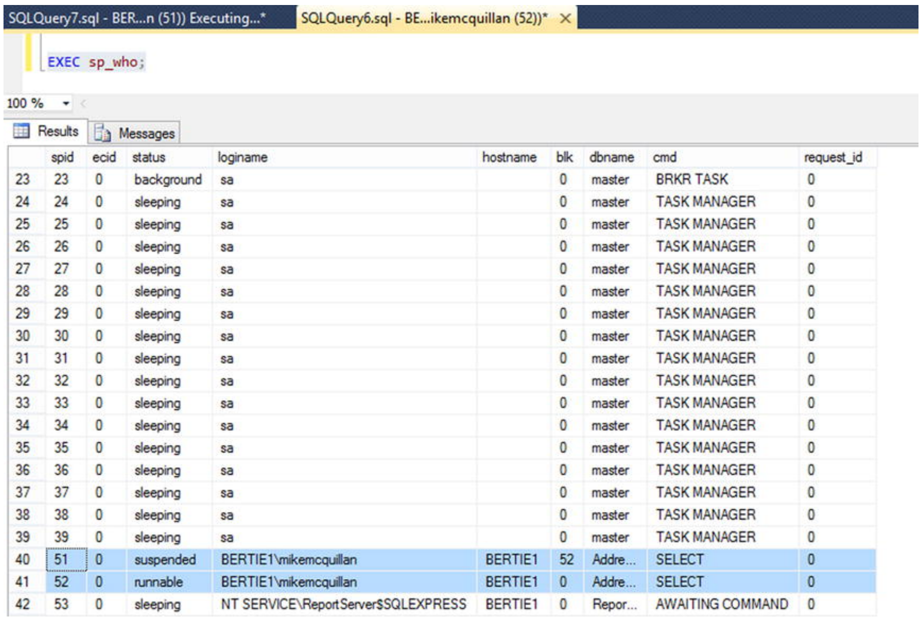
Figure 15-4. *Proving a transaction is still running with* `@@TRANCOUNT`

sp_who

As well as global system variables, there are also several system stored procedures and functions we can use t make our lives easier. System stored procedures begin with `sp_`. `sp_who` is a system stored procedure that can tell us who is holding locks (a lock prevents other users from accessing a particular resource in SQL Server while you are using it; e.g., a table). Replace `SELECT @@TRANCOUNT` with this:

```
EXEC sp_who;
```

Highlight this and run it. A lot of results should appear. Scroll to the bottom as shown in Figure 15-5. You are looking for a row that does not have a value of **0** in the **blk** column.


SQLQuery7.sql - BER...n (51)) Executing...*
SQLQuery6.sql - BE...ikemcquillan (52))*
100 %
Results
Messages

```
EXEC sp_who;
```

	spid	ecid	status	loginame	hostname	blk	dbname	cmd	request_id
23	23	0	background	sa		0	master	BRKR TASK	0
24	24	0	sleeping	sa		0	master	TASK MANAGER	0
25	25	0	sleeping	sa		0	master	TASK MANAGER	0
26	26	0	sleeping	sa		0	master	TASK MANAGER	0
27	27	0	sleeping	sa		0	master	TASK MANAGER	0
28	28	0	sleeping	sa		0	master	TASK MANAGER	0
29	29	0	sleeping	sa		0	master	TASK MANAGER	0
30	30	0	sleeping	sa		0	master	TASK MANAGER	0
31	31	0	sleeping	sa		0	master	TASK MANAGER	0
32	32	0	sleeping	sa		0	master	TASK MANAGER	0
33	33	0	sleeping	sa		0	master	TASK MANAGER	0
34	34	0	sleeping	sa		0	master	TASK MANAGER	0
35	35	0	sleeping	sa		0	master	TASK MANAGER	0
36	36	0	sleeping	sa		0	master	TASK MANAGER	0
37	37	0	sleeping	sa		0	master	TASK MANAGER	0
38	38	0	sleeping	sa		0	master	TASK MANAGER	0
39	39	0	sleeping	sa		0	master	TASK MANAGER	0
40	51	0	suspended	BERTIE1\mikemcquillan	BERTIE1	52	Addre...	SELECT	0
41	52	0	runnable	BERTIE1\mikemcquillan	BERTIE1	0	Addre...	SELECT	0
42	53	0	sleeping	NT SERVICE\ReportServer$SQLEXPRESS	BERTIE1	0	Repor...	AWAITING COMMAND	0

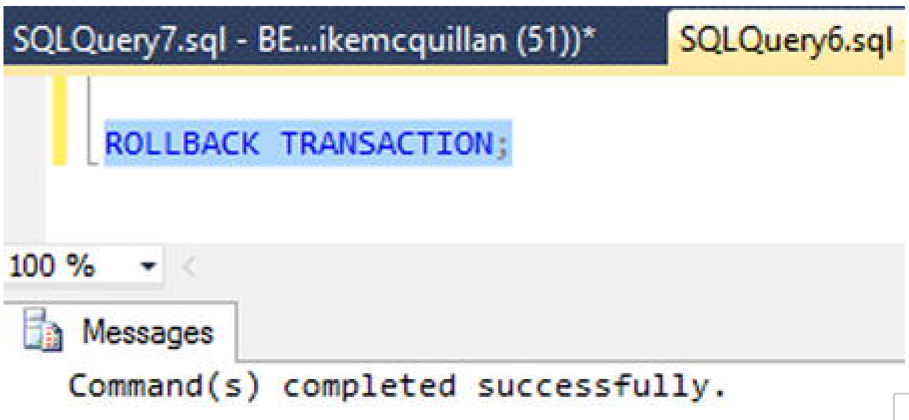
Figure 15-5. *Finding out who is causing resource blocking*

The first column in Figure 15-5, `spid`, represents a process ID. Each connection has one of these, allowing each connection to be uniquely identified. The `status` column tells us where each process ID is at—we can see `spid` 51 is currently suspended. This means it is waiting for something else to happen before it can continue processing. The something else is made evident by the `blk` column—the blocked by column. `spid` 52 is blocking `spid` 51.

Rolling Back the Bad Transaction

To resolve this situation, type this code into the `BEGIN TRANSACTION` window, highlight it, and run it (Figure 15-6).

```
ROLLBACK TRANSACTION;
```

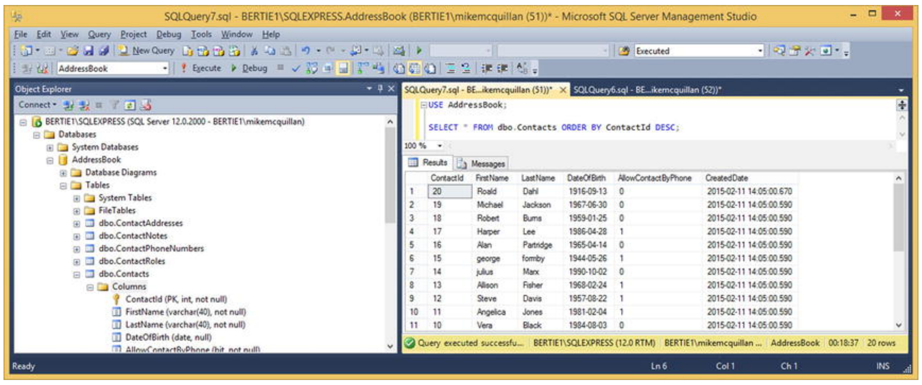


```
ROLLBACK TRANSACTION;
```

```
Command(s) completed successfully.
```

Figure 15-6. *Rolling back the open transaction*

As soon as you run this, the `SELECT` window should stop executing. Return to the `SELECT` window and it should now be displaying the contents of the `Contacts` table (Figure 15-7).



SQLQuery7.sql - BERTIE1\SQLEXPRESS.AddressBook (BERTIE1\mikemcquillan (51))* - Microsoft SQL Server Management Studio

```
USE AddressBook;

SELECT * FROM dbo.Contacts ORDER BY ContactId DESC;
```

	ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
1	20	Roald	Dahl	1916-09-13	0	2015-02-11 14:05:00.670
2	19	Michael	Jackson	1967-06-30	0	2015-02-11 14:05:00.590
3	18	Robert	Burns	1959-01-25	0	2015-02-11 14:05:00.590
4	17	Harper	Lee	1986-04-28	1	2015-02-11 14:05:00.590
5	16	Alan	Partridge	1965-04-14	0	2015-02-11 14:05:00.590
6	15	george	formby	1944-05-26	1	2015-02-11 14:05:00.590
7	14	julius	Marx	1990-10-02	0	2015-02-11 14:05:00.590
8	13	Allison	Fisher	1968-02-24	1	2015-02-11 14:05:00.590
9	12	Steve	Davis	1957-08-22	1	2015-02-11 14:05:00.590
10	11	Angelica	Jones	1981-02-04	1	2015-02-11 14:05:00.590
11	10	Vera	Black	1984-08-03	0	2015-02-11 14:05:00.590

Query executed successfu... | BERTIE1\SQLEXPRESS (12.0 RTM) | BERTIE1\mikemcquillan ... | AddressBook | 00:18:37 | 20 rows

Figure 15-7. *The SELECT statement finally completes!*

Look in the bottom right-hand corner of Figure 15-7—it took 18 minutes and 37 seconds to execute that `SELECT` statement! This is the time it took me to return to the other window and roll back the transaction.

The new contact we added—`ContactId` 21—has disappeared. This is because we rolled it back. It no longer exists. Actually, it never did exist; it only existed in the window in which we started the transaction. The row was not committed to the table, so it was never properly saved to the database.

Let this be a sobering lesson to you regarding badly written transactions! With great power comes great responsibility. When writing a transaction, always, and I do mean always, make sure it can either commit or roll back, no matter which path the code takes. You won't appreciate a call at 3:00 in the morning telling you some parts of the database cannot be accessed!

Committing Transactions

Let's return to our `BEGIN TRANSACTION` script. We'll finish this off so it works correctly. Remove the `ROLLBACK TRANSACTION`, `EXEC sp_who`, and any other lines left over from our earlier demonstration, so the script looks as it did when we first typed it in.

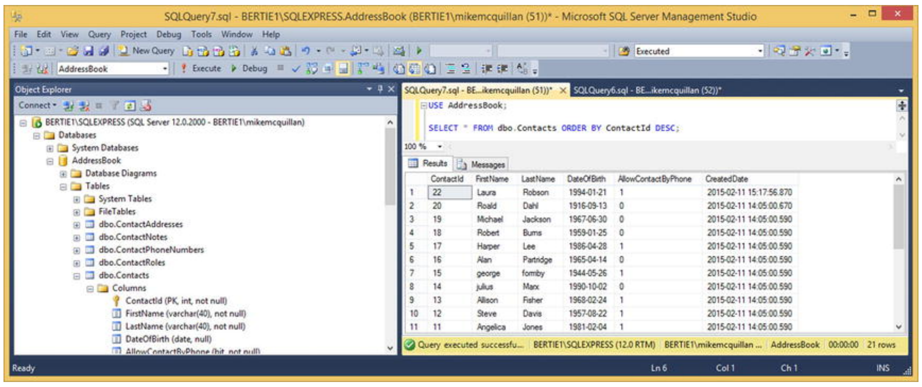
So far, this script inserts into the `Contacts` table. It also needs to insert into the `ContactVerificationDetails` table, so add that `INSERT` statement, and finish off by adding a `COMMIT TRANSACTION` statement. Here's the full script:

```
USE AddressBook;

DECLARE @ContactId INT;

BEGIN TRANSACTION;

INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone) VALUES('Laura', 'Ro

SELECT @ContactId = SCOPE_IDENTITY();

PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));

INSERT INTO dbo.ContactVerificationDetails(ContactId, DrivingLicenseNumber, ContactVerified) VALUES

COMMIT TRANSACTION;
```

Now we have a fully operational transaction. We add a record to `Contacts`, obtain the `ContactId`, and then insert a matching record into `ContactVerificationDetails`. Run this now and you should see some positive messages:

```
(1 row(s) affected)
Inserted contact ID: 22

(1 row(s) affected)
```

Pop over to the other window containing your `SELECT` statement. This time it should run without issue, and the new contact should be present in the results too, as `ContactId` 22 (you can see this contact in the results of Figure 15-8).



```
USE AddressBook;

SELECT * FROM dbo.Contacts ORDER BY ContactId DESC;
```

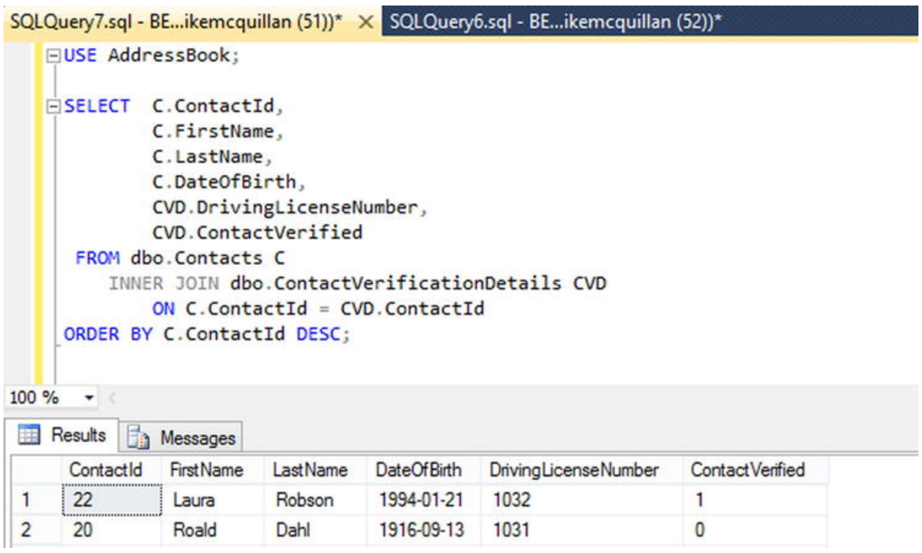
	ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
1	22	Laura	Robson	1994-01-21	1	2015-02-11 15:17:56.870
2	20	Roald	Dahl	1916-09-13	0	2015-02-11 14:05:00.670
3	19	Michael	Jackson	1967-06-30	0	2015-02-11 14:05:00.590
4	18	Robert	Burns	1959-01-25	0	2015-02-11 14:05:00.590
5	17	Harper	Lee	1986-04-28	1	2015-02-11 14:05:00.590
6	16	Alan	Partridge	1965-04-14	0	2015-02-11 14:05:00.590
7	15	george	formby	1944-05-26	1	2015-02-11 14:05:00.590
8	14	julius	Marx	1990-10-02	0	2015-02-11 14:05:00.590
9	13	Allison	Fisher	1968-02-24	1	2015-02-11 14:05:00.590
10	12	Steve	Davis	1957-08-22	1	2015-02-11 14:05:00.590
11	11	Angelica	Jones	1981-02-04	1	2015-02-11 14:05:00.590

Query executed successfu... BERTIE1\SQLEXPRESS (12.0 RTM) BERTIE1\mikemcquillan ... AddressBook 00:00:00 21 rows

Figure 15-8. The new contact is finally created

The query took less than one second to run. If you modify the query to join to the `ContactVerificationDetails` table, the `INNER JOIN` will guarantee that `ContactId` 22 does indeed have a matching `ContactVerificationDetails` record (see Figure 15-9).

```
USE AddressBook;

SELECT C.ContactId, C.FirstName, C.LastName, C.DateOfBirth, CVD.DrivingLicenseNumber, CVD.ContactVe
```



```
USE AddressBook;

SELECT  C.ContactId,
        C.FirstName,
        C.LastName,
        C.DateOfBirth,
        CVD.DrivingLicenseNumber,
        CVD.ContactVerified
  FROM dbo.Contacts C
     INNER JOIN dbo.ContactVerificationDetails CVD
        ON C.ContactId = CVD.ContactId
ORDER BY C.ContactId DESC;
```

	ContactId	FirstName	LastName	DateOfBirth	DrivingLicenseNumber	ContactVerified
1	22	Laura	Robson	1994-01-21	1032	1
2	20	Roald	Dahl	1916-09-13	1031	0

Figure 15-9. Proving records in both tables were created

As Figure 15-9 proves to us, a successful insert occurred. Thanks to our transaction, it wasn't possible to add a `Contacts` record without a corresponding `ContactVerificationDetails` record.

But wait! We know things work if everything runs as expected. What happens if something goes wrong? We'll need a rollback!

Rolling Back Transactions

We used the `ROLLBACK TRANSACTION` statement earlier to fix our locking problem. Now it's time to be a bi more proactive and put rollback at the heart of our script. We'll add a second contact. Change our `INSERT` script so it looks like this:

```
USE AddressBook;

DECLARE @ContactId INT,
@InsertSuccessful BIT = 0;

BEGIN TRANSACTION;

INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone) VALUES('Bryan', 'Fe

SELECT @ContactId = SCOPE_IDENTITY();

PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));

IF (COALESCE(@ContactId, 0) != 0)
BEGIN
INSERT INTO dbo.ContactVerificationDetails(ContactId, DrivingLicenseNumber, ContactVerified) VALUES

SELECT @InsertSuccessful = 1;
END;

IF (@InsertSuccessful = 1)
BEGIN
COMMIT TRANSACTION;
PRINT 'Transaction committed successfully.';
END;
ELSE
BEGIN
ROLLBACK TRANSACTION;
PRINT 'Transaction rolled back. No changes made.';
END;
```

There are quite a few changes here. We've added a new variable, to hold whether our insert was successful or not. We then perform the insert into Contacts, inserting Roxy Music's Bryan Ferry instead of tennis's Laura Robson.

Now there is some new code. We check if a valid `ContactId` value has been assigned to the `@ContactId` variable. If the `@ContactId` value is not zero or null, it is deemed to be a valid value and the insert into `ContactVerificationDetails` can occur. The code then sets the `@InsertSuccessful` flag to 1, denoting all inserts have succeeded. This is important, as it dictates whether the commit or rollback will occur. By default the `@InsertSuccessful` variable is set to 0, so we are assuming failure from the beginning of the script—we only change the variable's value if the code succeeds.

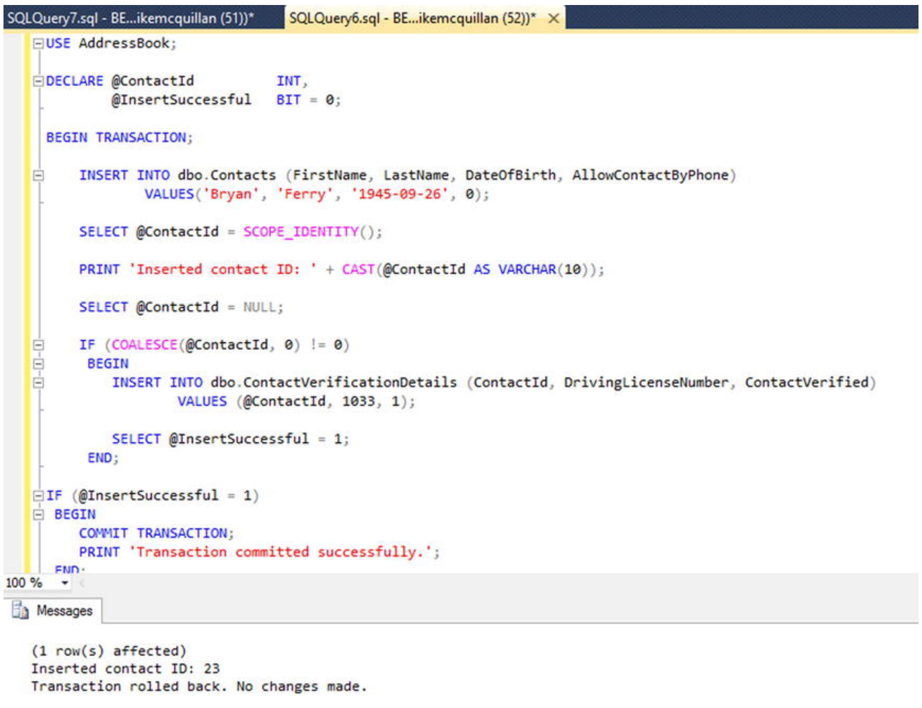
The last block of code executes either the `COMMIT` or `ROLLBACK` statement. The `@InsertSuccessful` variable is checked. If this is set to 1 the changes will be committed and the transaction has succeeded. The data will be available in the tables. If `@InsertSuccessful` failed, the transaction will be rolled back and no changes will be made.

Let's see the rollback kick in. Between the `PRINT` and `IF (COALESCE)` statements, add this line:

```
SELECT @ContactId = NULL;
```

We're simulating a failure with this line. By setting `@ContactId` to `NULL` the `COALESCE` check will fail, no insert will be made into `ContactVerificationDetails`, and the `@InsertSuccessful` flag will remain set to 0.

Run this script and you should see a message telling you the changes were rolled back, just like the one in Figure 15-10.



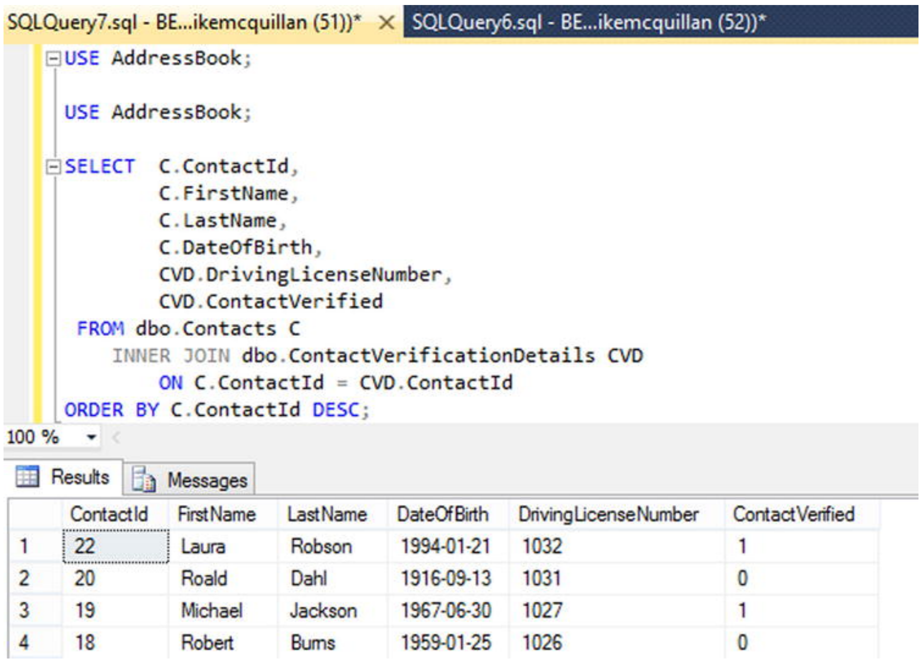
SQLQuery7.sql - BE...ikemcquillan (51))*
SQLQuery6.sql - BE...ikemcquillan (52))*
100 %
Messages

```
USE AddressBook;

DECLARE @ContactId          INT,
        @InsertSuccessful   BIT = 0;

BEGIN TRANSACTION;

    INSERT INTO dbo.Contacts (FirstName, LastName, DateOfBirth, AllowContactByPhone)
            VALUES('Bryan', 'Ferry', '1945-09-26', 0);

    SELECT @ContactId = SCOPE_IDENTITY();

    PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));

    SELECT @ContactId = NULL;

    IF (COALESCE(@ContactId, 0) != 0)
     BEGIN
        INSERT INTO dbo.ContactVerificationDetails (ContactId, DrivingLicenseNumber, ContactVerified)
                VALUES (@ContactId, 1033, 1);

        SELECT @InsertSuccessful = 1;
     END;

IF (@InsertSuccessful = 1)
 BEGIN
    COMMIT TRANSACTION;
    PRINT 'Transaction committed successfully.';
 END;
```

```
(1 row(s) affected)
Inserted contact ID: 23
Transaction rolled back. No changes made.
```

Figure 15-10. A transaction that has just been rolled back

It doesn't matter how many times you run the script; no data will be inserted. Visit the `SELECT` statement in the other window and run it. Laura Robson, `ContactId` 22, will still be the last contact added. Check out the results in Figure 15-11 for evidence.



SQLQuery7.sql - BE...ikemcquillan (51))*
SQLQuery6.sql - BE...ikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

USE AddressBook;

SELECT  C.ContactId,
        C.FirstName,
        C.LastName,
        C.DateOfBirth,
        CVD.DrivingLicenseNumber,
        CVD.ContactVerified
 FROM dbo.Contacts C
    INNER JOIN dbo.ContactVerificationDetails CVD
        ON C.ContactId = CVD.ContactId
ORDER BY C.ContactId DESC;
```

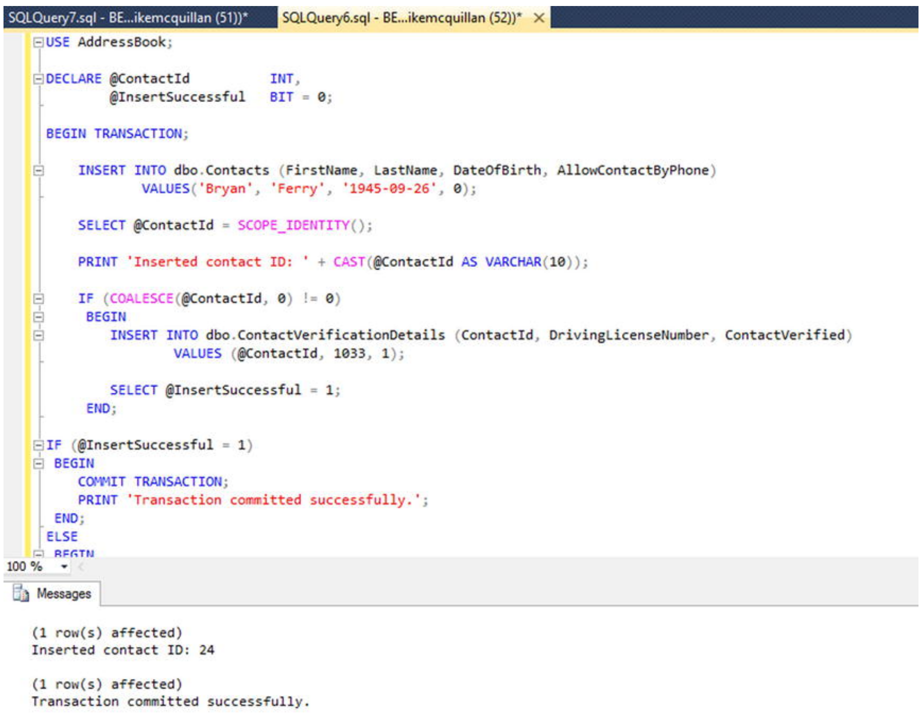
	ContactId	FirstName	LastName	DateOfBirth	DrivingLicenseNumber	ContactVerified
1	22	Laura	Robson	1994-01-21	1032	1
2	20	Roald	Dahl	1916-09-13	1031	0
3	19	Michael	Jackson	1967-06-30	1027	1
4	18	Robert	Burns	1959-01-25	1026	0

Figure 15-11. Laura Robson is still the latest contact

Return to the insert script and remove the extra line we added:

```
SELECT @ContactId = NULL;
```

Run the script again. This time, you should see the success message shown in Figure 15-12.



SQLQuery7.sql - BE...ikemcquillan (51))* SQLQuery6.sql - BE...ikemcquillan (52))*

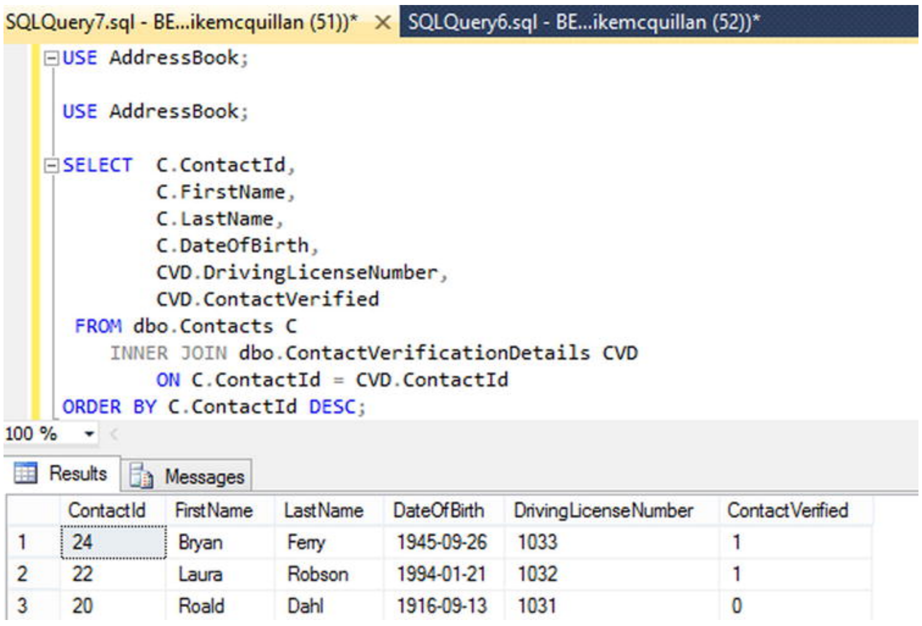
```
USE AddressBook;

DECLARE @ContactId          INT,
        @InsertSuccessful   BIT = 0;

BEGIN TRANSACTION;

    INSERT INTO dbo.Contacts (FirstName, LastName, DateOfBirth, AllowContactByPhone)
            VALUES('Bryan', 'Ferry', '1945-09-26', 0);

    SELECT @ContactId = SCOPE_IDENTITY();

    PRINT 'Inserted contact ID: ' + CAST(@ContactId AS VARCHAR(10));

    IF (COALESCE(@ContactId, 0) != 0)
     BEGIN
        INSERT INTO dbo.ContactVerificationDetails (ContactId, DrivingLicenseNumber, ContactVerified)
                VALUES (@ContactId, 1033, 1);

        SELECT @InsertSuccessful = 1;
     END;

IF (@InsertSuccessful = 1)
 BEGIN
    COMMIT TRANSACTION;
    PRINT 'Transaction committed successfully.';
 END;
ELSE
 BEGIN
```

Messages

```
(1 row(s) affected)
Inserted contact ID: 24

(1 row(s) affected)
Transaction committed successfully.
```

Figure 15-12. Successfully inserting within a committed transaction

I love it when a plan comes together! Visit the `SELECT` statement window and run the statement. Figure 15-13 shows that Bryan Ferry has joined our contact list, and he did it in an ACID-compliant manner, too. Well done, Mr. Ferry.



SQLQuery7.sql - BE...ikemcquillan (51))* SQLQuery6.sql - BE...ikemcquillan (52))*

```
USE AddressBook;

USE AddressBook;

SELECT  C.ContactId,
        C.FirstName,
        C.LastName,
        C.DateOfBirth,
        CVD.DrivingLicenseNumber,
        CVD.ContactVerified
 FROM dbo.Contacts C
    INNER JOIN dbo.ContactVerificationDetails CVD
        ON C.ContactId = CVD.ContactId
ORDER BY C.ContactId DESC;
```

Results Messages

	ContactId	FirstName	LastName	DateOfBirth	DrivingLicenseNumber	ContactVerified
1	24	Bryan	Ferry	1945-09-26	1033	1
2	22	Laura	Robson	1994-01-21	1032	1
3	20	Roald	Dahl	1916-09-13	1031	0

Figure 15-13. Bryan Ferry puts in an appearance

General Transaction Rules

Now that we've covered the basics of transactions, there are some general rules you should be aware of before you start wrapping all of your code in transactions. It may seem from this chapter that you should do this, but that isn't the case.

Keep Transactions Short

The first and most crucial rule is to keep your transactions short. When you open a transaction, you are locking any objects involved in that transaction. Therefore, the shorter the transaction, the shorter the time SQL Server locks the objects, reducing contention in your database.

Limit Transactions to DML Statements

There isn't much point in wrapping a `SELECT` statement that takes 20 minutes to execute in a transaction. Sometimes you cannot avoid embedding `SELECT` statements in transactions, but generally you want to limit the contents of a transaction to DML statements.

Don't Be Afraid to Split Transactions Up

I've often come across developers who feel they must do everything in a single transaction. This is bunk! If you have some code that you feel will take too long to execute in a single transaction, don't worry about splitting it up into multiple transactions. There are various techniques you can use to roll back the results of previous transactions should a subsequent transaction fail—storing new or updated data in temp tables, for example, before committing it all at the end of the code block. If you assess your code carefully, you'll see there is seldom a need for a long-running transaction.

Summary

After the *Lord of the Rings*–style trek through the indexes chapter, this has felt more like the breezy style of *The Hobbit*. We've covered transactions in some detail, and you should now be happy with terms like `BEGIN`, `ROLLBACK`, and `COMMIT`. It's important that users have faith in the data held within your database, and transactions are a major part of providing that faith.

We're going to move on to functions now, but it's worth saving the code we've written in this chapter—we'll be revisiting it when we look at stored procedures.

PREV
Chapter 14 : Indexes

NEXT
Chapter 16 : Functions