

## Lab 3 – More on T-SQL

### Content:

- Learn and practice data manipulation T-SQL commands
- Study and practice Transaction in database

### Learning outcome:

- Practical knowledge of data manipulation SQL commands
- How to manage Transaction in database

## Part 1: Updating data

Ensuring that you update the right data at the right time is crucial to maintaining data integrity. You will find that when updating data, and also when removing or inserting data, it is best to group this work as a single, logical unit, called a transaction, thereby ensuring that if an error does occur, it is still possible to return the data back to its original state. This section describes how a transaction works and how to incorporate transactions within your code. When looking at transactions, we will only be taking an overview of them. We will look at the basics of a transaction and how it can affect the data.

Deleting data can take one of two forms. The first is where a deletion of the data is logged in the transaction log. This means that if there is a failure of some sort, the deletion can be backed out. The second is where the deletion of the data is minimally logged. Knowing when to use each of these actions can improve performance of deletions.

### The UPDATE Statement

The UPDATE statement will update columns of information on rows within a single table returned from a query that can include selection and join criteria. The syntax of the UPDATE statement has similarities to the SELECT statement, which makes sense, as it has to look for specific rows to update, just as the SELECT statement looks for rows to retrieve.

```
UPDATE
    [ TOP ( expression ) [ PERCENT ] ]
    [[ server_name . database_name . schema_name .
    | database_name .[ schema_name ] .
    | schema_name .]
    table_or_viewname
    SET
        { column_name = { expression | DEFAULT | NULL }
        | column_name { .WRITE ( expression , @Offset , @Length ) }
        | @variable = expression
```

```
| @variable = column = expression [ ,...n ]  
} [ ,...n ]  
[FROM { <table_source> } [ ,...n ] ]  
[ WHERE { <search_condition> ]
```

The first set of options we know from the SELECT statement. The **tablename** clause is simply the name of the table on which to perform the UPDATE. Moving on to the next line of the syntax, we reach the SET clause. It is in this clause that any updates to a column take place. One or more columns can be updated at any one time, but each column to be updated must be separated by a comma.

When updating a column, there are four choices that can be made for data updates. This can be through a direct value setting; a section of a value setting providing that the recipient column is varchar, nvarchar, or varbinary; the value from a variable; or a value from another column or even from another table. We can even include mathematical functions or variable manipulations in the right-hand clause, include concatenated columns, or manipulate the contents through STRING, DATE, or any other function. Providing that the end result sees the left-hand side having the same data type as the right-hand side, the update will then be successful. As a result, we cannot place a character value into a numeric data type field without converting the character to a numeric value. If we are updating a column with a value from another column, the only value that it is possible to use is the value from the same row of information in another column, provided this column has an appropriate data type. When we say “same row,” remember that when tables are joined together, this means that values from these joined tables can also be used as they are within the same row of information. Also, the expression could be the result of a subquery.

The FROM table source clause will define the table(s) used to find the data to perform the update on the table defined next to the UPDATE statement. Like SELECT statements, it is possible to create JOIN statements; however, you must define the table you are updating within the FROM clause.

Finally, the WHERE condition is exactly as in the SELECT statement, and can be used in exactly the same way. Note that omitting the WHERE clause will mean the UPDATE statement will affect every row in the table.

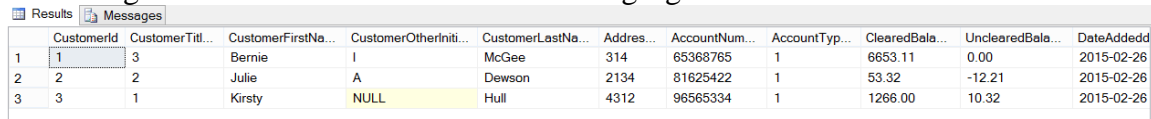
## **Practice: Updating Data**

First of all, it is necessary to run ApressFinancial\_script file to prepare the database.

1. Ensure that Query Editor is running and that you are within the ApressFinancial database. In the Query Editor pane, enter the following SQL code.

```
SELECT * FROM CustomerDetails.Customers
```

2. Execute the code using F5, or the execute button on the toolbar. You should then see something like the results shown in the following figure.



	CustomerId	CustomerTitle	CustomerFirstName	CustomerOtherInitials	CustomerLastName	Address	AccountNumber	AccountType	ClearedBalance	UnclearedBalance	DateAdded
1	1	3	Bernie	I	McGee	314	65368765	1	6653.11	0.00	2015-02-26
2	2	2	Julie	A	Dewson	2134	81625422	1	53.32	-12.21	2015-02-26
3	3	1	Kirsty	NULL	Hull	4312	96565334	1	1266.00	10.32	2015-02-26

3. Ensure that Query Editor is running and that you are logged in with an account that can perform updates. In the Query Editor pane, enter the following UPDATE statement:

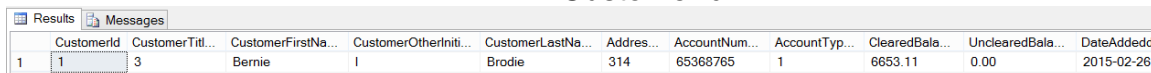
```
UPDATE CustomerDetails.Customers  
SET CustomerLastName = 'Brodie'  
WHERE CustomerId = 1
```

4. It is as simple as that! Now that the code is entered, execute the code, and you should then see a message like this:

(1 row(s) affected)

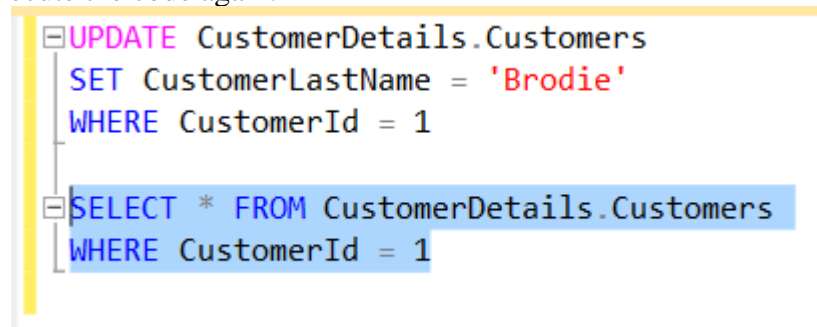
5. Now enter a SELECT statement to check that Bernie McGee is now Bernie Brodie. For your convenience, here's the statement, and the results are shown in the following figure:

```
SELECT * FROM CustomerDetails.Customers  
WHERE CustomerId = 1
```



	CustomerId	CustomerTitle	CustomerFirstName	CustomerOtherInitials	CustomerLastName	Address	AccountNumber	AccountType	ClearedBalance	UnclearedBalance	DateAdded
1	1	3	Bernie	I	Brodie	314	65368765	1	6653.11	0.00	2015-02-26

6. Now here's a little trick that you should know, if you haven't stumbled across it already. If you check out the following figure, you will see that the UPDATE code is still in the Query Editor pane, as is the SELECT statement. No, we aren't going to perform the UPDATE again! If you highlight the line with the SELECT statement by holding down the left mouse button and dragging the mouse, only the highlighted code will run when you execute the code again.



```
UPDATE CustomerDetails.Customers  
SET CustomerLastName = 'Brodie'  
WHERE CustomerId = 1  
  
SELECT * FROM CustomerDetails.Customers  
WHERE CustomerId = 1
```

7. It is also possible to update data using information from another column within the table, or with the value from a variable. This next example will demonstrate how to update a row of information using the value within a variable, and a column from the same table. Notice how although the row will be found using the CustomerLastName column, the UPDATE statement is also updating that column with a new value. Enter the following code and then execute it:

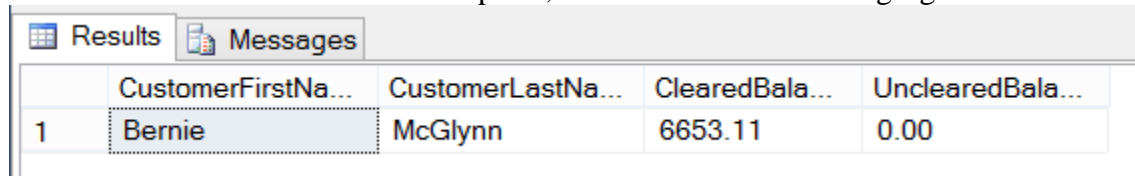
```
DECLARE @ValueToUpdate VARCHAR(30)
SET @ValueToUpdate = 'McGlynn'
UPDATE CustomerDetails.Customers
    SET CustomerLastName = @ValueToUpdate,
        ClearedBalance = ClearedBalance + UnclearedBalance ,
        UnclearedBalance = 0
WHERE CustomerLastName = 'Brodie'
```

8. You should then see the following output:  
(1 row(s) affected)

9. Now to check what has happened. You may be thinking that the update has not happened because you are altering the column that is being used to find the row, but this is not so. The row is found, the update occurs, and then the row is written back to the table. Once the row is retrieved for update, there is no need for that value to be kept. Just check that the update occurred by entering and executing the following code:

```
SELECT CustomerFirstName, CustomerLastName,
    ClearedBalance, UnclearedBalance
FROM CustomerDetails.Customers
WHERE CustomerId = 1
```

You should now see the alteration in place, as shown in the following figure.



	CustomerFirstNa...	CustomerLastNa...	ClearedBala...	UnclearedBala...
1	Bernie	McGlynn	6653.11	0.00

10. Now let's move on to updating columns in which the data types don't match. SQL server does a pretty good job when it can ensure the update occurs, and these following examples will demonstrate how well SSE copes with updating an integer data type with a value in a varchar data type. The first example will demonstrate where a varchar value will successfully update a column defined as integer. Enter the following code:

```
DECLARE @WrongDataType VARCHAR(20)
SET @WrongDataType = '4311.22'
UPDATE CustomerDetails.Customers
    SET ClearedBalance = @WrongDataType
WHERE CustomerId = 1
```

11. Execute the code; you should see the following message when you do:

(1 row(s) affected)

12. The value 4311.22 has been placed into the ClearedBalance column for CustomerId 1. SSE has performed an internal data conversion (known as an implicit data type conversion) and has come up with a money data type from the value within varchar, as this is what the column expects, and therefore can successfully update the column. Here is the output as proof:

```
SELECT CustomerFirstName, CustomerLastName,  
       ClearedBalance, UnclearedBalance  
FROM CustomerDetails.Customers  
WHERE CustomerId = 1
```

The following shows the results of updating the column.

Results		Messages		
	CustomerFirstNa...	CustomerLastNa...	ClearedBala...	UnclearedBala...
1	Bernie	McGlynn	4311.22	0.00

13. However, in this next example, the data type that SQL Server will come up with is a numeric data type. When we try to alter an integer-based data type, bigint, with this value, which we can find in the AddressId column, the UPDATE does not take place. Enter the following code:

```
DECLARE @WrongDataType VARCHAR(20)  
SET @WrongDataType = '2.0'  
UPDATE CustomerDetails.Customers  
SET AddressId = @WrongDataType  
WHERE CustomerId = 1
```

14. Now execute the code. Notice when we do that SQL Server generates an error message informing us of the problem. Hence, never leave data conversions to SQL Server to perform. Try to get the same data type updating the same data type.

Msg 8114, Level 16, State 5, Line 3  
Error converting data type varchar to bigint.

Updating data can be very straightforward, as the preceding examples have demonstrated. Where at all possible, either use a unique identifier, for example, the CustomerId, when trying to find a customer, rather than a name. There can be multiple rows for the same name or other type of criteria, but by using the unique identifier, you can be sure of using the right row every time. To place this in a production scenario, we would have a Windows-based graphical system that would allow you to find details of customers by their name, address, or account number. Once you found the right customer, instead of keeping those details to find other related rows, keep a record of the unique identifier value instead.

Getting back to the UPDATE statement and how it works, first of all SQL Server will filter out from the table the first row that meets the criteria of the WHERE statement. The

data modifications are then made, and SQL Server moves on to try to find the second row matching the WHERE statement. This process is repeated until all the rows that meet the WHERE condition are modified. Therefore, if using a unique identifier, SQL Server will only update one row, but the WHERE statement looks for rows that have a CustomerLastName of McGlynn, in which case multiple rows could be updated. So choose your row selection criteria for updates carefully.

## Part 2: Transaction

A transaction is a method through which developers can define a unit of work logically or physically that, when it completes, leaves the database in a consistent state. A transaction forms a single unit of work, which must pass the ACID test before it can be classified as a transaction. The ACID test is an acronym for Atomicity, Consistency, Isolation, and Durability:

**Atomicity:** In its simplest form, all data modifications within the transaction must be both accepted and inserted successfully into the database, or none of the modifications will be performed.

**Consistency:** Once the data has been successfully applied, or rolled back to the original state, all the data must remain in a consistent state, and the data must still maintain its integrity.

**Isolation:** Any modification in one transaction must be isolated from any modifications in any other transaction. Any transaction should see data from any other transaction either in its original state or once the second transaction has completed. It is impossible to see the data in an intermediate state.

**Durability:** Once a transaction has finished, all data modifications are in place and can only be modified by another transaction or unit of work. Any system failure (hardware or software) will not remove any changes applied.

Transactions within a database are a very important topic, but also one that requires a great deal of understanding. This chapter covers the basics of transactions only. To really do justice to this area, we would have to deal with some very complex and in-depth scenarios, covering all manner of areas such as triggers, nesting transactions, and transaction logging, which is beyond the scope of this book.

A transaction can be placed around any data manipulation, whether it is an update, insertion, or deletion, and can cater to one row or many rows, and also many different statements. There is no need to place a transaction around a SELECT statement unless you are doing a SELECT...INTO, which is of course modifying data. This is because a transaction is only required when data manipulation occurs such that changes will either be committed to the table or discarded. A transaction could cover several UPDATE, DELETE, or INSERT statements, or indeed a mixture of all three. However, there is one very large warning that goes with using transactions.

A deadlock is where two separate data manipulations, in different transactions, are being performed at the same time. However, each transaction is waiting for the other to finish the update before it can complete its update. Neither manipulation can be completed because each is waiting for the other to finish. A deadlock occurs, and it can (and will) lock the tables and database in question. So, for example, transaction 1 is updating the

customers table followed by the customer transactions table. Transaction 2 is updating the customer transactions table followed by the customers table. A lock would be placed on the customers table while those updates were being done by transaction 1. A lock would be placed on the customer transactions table by transaction 2. Transaction 1 could not proceed because of the lock by transaction 2, and transaction 2 could not proceed due to the lock created by transaction 1. Both transactions are “stuck.” So it is crucial to keep the order of table updates the same, especially where both could be running at the same time.

It is also advisable to keep transactions as small, and as short, as possible, and under no circumstances hold onto a lock for more than a few seconds. We can do this by keeping the processing within a transaction to as few lines of code as possible, and then either roll back (that is, cancel) or commit the transaction to the database as quickly as possible within code. With every second that you hold a lock through a transaction, you are increasing the potential of trouble happening. In a production environment, with every passing millisecond that you hold onto a piece of information through a lock, you are increasing the chances of someone else trying to modify the same piece of information at the same time and the possibility of the problems that would then arise.

There are two parts that make up a transaction, the start of the transaction and the end of the transaction, where you decide whether you want to commit the changes or revert back to the original state. We will now look at the definition of the start of the transaction, and then the T-SQL statements required to commit or roll back the transaction. The basis of this section is that only one transaction is in place, and that you have no nested transactions. Nested transactions are much more complex and should only really be dealt with once you are proficient with SSE. The statements we are going through in the upcoming text assume a single transaction; the COMMIT TRAN section changes slightly when the transaction is nested.

### **BEGIN TRAN**

The T-SQL statement BEGIN TRAN denotes the start of the transaction processing. From this point on, until the transaction is ended with either COMMIT TRAN or ROLLBACK TRAN, any data modification statements will form part of the transaction. It is also possible to suffix the BEGIN TRAN statement with a name of up to 32 characters in length. If you name your transaction, it is not necessary to use the name when issuing a ROLLBACK TRAN or a COMMIT TRAN statement. The name is there for clarity of the code only.

### **COMMIT TRAN**

The COMMIT TRAN statement will commit the data modifications to the database permanently, and there will be no going back once this statement is executed. This function should only be executed when all changes to the database are ready to be committed.



## **ROLLBACK TRAN**

If you wish to remove all the database changes that have been completed since the beginning of the transaction, say, for example, because an error had occurred, then you could issue a ROLLBACK TRAN statement.

So, if you were to start a transaction with BEGIN TRAN and then issue an INSERT that succeeds, and then perhaps an UPDATE that fails, you could issue a ROLLBACK TRAN to roll back the transaction as a whole. As a result, you roll back not only the UPDATE changes, but also, because they form part of the same transaction, the changes made by the INSERT, even though that particular operation was successful.

To reiterate, keep transactions small and short. Never leave a session with an open transaction by having a BEGIN TRAN with no COMMIT TRAN or ROLLBACK TRAN. Ensure that you do not cause a deadly embrace.

If you issue a BEGIN TRAN, then you MUST issue a COMMIT TRAN or ROLLBACK TRAN transaction as quickly as possible; otherwise, the transaction will stay around until the connection is terminated.

## **Locking Data**

The whole area of locking data, how locks are held, and how to avoid problems with them, is a very large complex area and not for the fainthearted. However, it is necessary to be aware of locks, and at least have a small amount of background knowledge on them so that when you design your queries, you stand a chance of avoiding problems.

The basis of locking is to allow one transaction to update data, knowing that if it has to roll back any changes, no other transaction has modified the data since the first transaction did.

To explain this with an example, if you have a transaction that updates the CustomerDetails.Customers table, and then moves on to update the TransactionDetails.Transactions table, but hits a problem when updating the TransactionDetails.Transactions table, the transaction must be safe in the knowledge that it is only rolling back the changes it made, and not changes by another transaction. Therefore, until all the table updates within the transaction are either successfully completed or have been rolled back, the transaction will keep hold of any data inserted, modified, or deleted.

However, one problem with this approach is that SQL Server may not just hold the data that the transaction has modified. Keeping a lock on the data that has just been modified is called row-level locking. On the other hand, SQL Server may be set up to lock the database, which is known as database-level locking, found in areas such as backups and restores. The other levels in between are row, page, and table locking, and so you could lock a large resource, depending on the task being performed.

This is about as deep as I will take this discussion on locks, so as not to add any confusion or create a problematic situation. Dealing with locks is handled automatically by SSE, but it is possible to make locking more efficient by developing an effective understanding of the subject, and then customizing the locks within your transactions.

### **Practice: Updating Data Using Transactions**

Now, what if, in the first update query of this chapter, we had made a mistake or an error occurred? For example, say we chose the wrong customer, or even worse, omitted the WHERE statement, and therefore all the rows were updated. These are unusual errors, but quite possible. More common errors could result from where more than one data modification has to take place and succeed, and the first one succeeds but a subsequent modification fails. By using a transaction, we would have had the chance to correct any mistakes easily, and could then revert to a consistent state. Of course, this next example is nice and simple, but by working through it, the subject of transactions will hopefully become a little easier to understand and appreciate.

1. Make sure Query Editor is running for this first example, which will demonstrate COMMIT TRAN in action. There should be no difference from an UPDATE without any transaction processing, as it will execute and update the data successfully. However, this should prove to be a valuable exercise, as it will also demonstrate the naming of a transaction. Enter the following code:

```
SELECT 'Before',ShareId,ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE ShareId = 2
BEGIN TRAN ShareUpd
UPDATE ShareDetails.Shares
SET CurrentPrice = CurrentPrice * 1.1
WHERE ShareId = 2
COMMIT TRAN
SELECT 'After',ShareId,ShareDesc,CurrentPrice
FROM ShareDetails.Shares
WHERE ShareId = 2
```

Notice in the preceding code that the COMMIT TRAN does not use the name associated with the BEGIN TRAN. The label after the BEGIN TRAN is simply that, a label, and it performs no functionality. It is therefore not necessary to then link up with a similarly labeled COMMIT TRAN.

2. Execute the code. The following figure shows the results, which list out the ShareDetails.Shares table before and after the transaction.

Results		Messages		
	(No column na...	Shar...	ShareDesc	CurrentPri...
1	Before	2	ACME'S HOMEBAKE COOKIES INC	2.34125
	(No column na...	Shar...	ShareDesc	CurrentPri...
1	After	2	ACME'S HOMEBAKE COOKIES INC	2.57538

3. We are now going to work through a ROLLBACK TRAN. We will take this one stage at a time so that you fully understand and follow the processes involved. Note in the following code that the WHERE statement has been commented out with --. By having the WHERE statement commented out, hopefully you'll have already guessed that every row in the ShareDetails.Shares table is going to be updated. The example needs you to execute all the code at once, so enter the following code into your Query Editor pane, and then execute it. Note we have three SELECT statements this time—before, during, and after the transaction processing.

```

SELECT 'Before',ShareId,ShareDesc,CurrentPrice
  FROM ShareDetails.Shares
  WHERE ShareId = 2
BEGIN TRAN ShareUpd
UPDATE ShareDetails.Shares
  SET CurrentPrice = CurrentPrice * 1.1
  WHERE ShareId = 2
SELECT 'Within the transaction',ShareId,ShareDesc,CurrentPrice
  FROM ShareDetails.Shares
ROLLBACK TRAN
SELECT 'After',ShareId,ShareDesc,CurrentPrice
  FROM ShareDetails.Shares
  WHERE ShareId = 2

```

4. The results, as you see in the following figure, show us exactly what has happened. Take a moment to look over these results. The first list shows the full set of rows in the ShareDetails.Shares table prior to our UPDATE. The middle rowset shows us the BEGIN transaction where we have updated every share, and the final listing shows the data restored back to its original state via a ROLLBACK TRAN.

Results Messages				
	(No column na...	Shar...	ShareDesc	CurrentPri...
1	Before	2	ACME'S HOMEBAKE COOKIES INC	2.57538
	(No column name)	Shar...	ShareDesc	CurrentPri...
1	Within the transaction	1	ACME'S HOMEBAKE COOKIES INC	2.34125
2	Within the transaction	2	ACME'S HOMEBAKE COOKIES INC	2.83292
	(No column na...	Shar...	ShareDesc	CurrentPri...
1	After	2	ACME'S HOMEBAKE COOKIES INC	2.57538

### Nested Transactions

Let's look at one last example before moving on. It is possible to nest transactions inside one another. This is not a complete coverage, as it can get very complex and messy if you involve save points, stored procedures, triggers, and so on. The aim of this section is to give you an understanding of the basic but crucial points of how nested transactions work.

Nested transactions can occur in a number of different scenarios. For example, you could have a transaction in one set of code in a stored procedure, which calls a second stored procedure that also has a transaction. We will look at a simpler scenario where we just keep the transactions in one set of code.

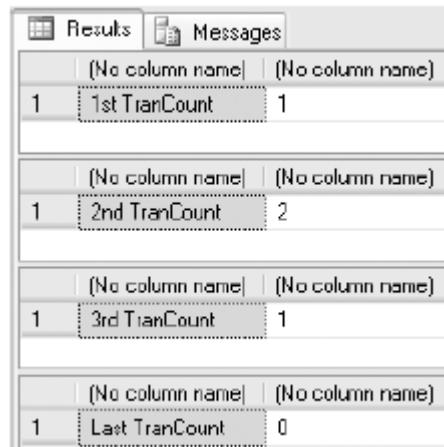
What you need to be clear about is how the ROLLBACK and COMMIT TRAN statements work in a nested transaction. First of all, let's see what we mean by nesting a simple transaction. The syntax is shown here, and you can see that two BEGIN TRAN statements occur before you get to a COMMIT or a ROLLBACK:

```
BEGIN TRAN
  Statements
  BEGIN TRAN
    Statements
  COMMIT|ROLLBACK TRAN
COMMIT|ROLLBACK TRAN
```

As each transaction commences, SQL Server adds 1 to a running count of transactions it holds in a system variable called @@TRANCOUNT. Therefore, as each BEGIN TRAN is executed, @@TRANCOUNT increases by 1. As each COMMIT TRAN is executed, @@TRANCOUNT decreases by 1. It is not until @@TRANCOUNT is at a value of 1 that you can actually commit the data to the database. The code that follows might help you to understand this a bit more.

Enter and execute this code and take a look at the output, which should resemble the following figure. The first BEGIN TRAN increases @@TRANCOUNT by 1, as does the second BEGIN TRAN. The first COMMIT TRAN marks the changes to be committed, but does not actually perform the changes because @@TRANCOUNT is 2. It simply creates the correct BEGIN/COMMIT TRAN nesting and reduces @@TRANCOUNT by 1. The second COMMIT TRAN will succeed and will commit the data, as @@TRANCOUNT is 1.

```
BEGIN TRAN ShareUpd
  SELECT '1st TranCount', @@TRANCOUNT
  BEGIN TRAN ShareUpd2
    SELECT '2nd TranCount', @@TRANCOUNT
  COMMIT TRAN ShareUpd2
  SELECT '3rd TranCount', @@TRANCOUNT
COMMIT TRAN -- It is at this point that data modifications will be committed
SELECT 'Last TranCount', @@TRANCOUNT
```



	(No column name)	(No column name)
1	1st TranCount	1
1	2nd TranCount	2
1	3rd TranCount	1
1	Last TranCount	0

If in the code there is a ROLLBACK TRAN, the data is immediately rolled back no matter where you are within the nesting, and @@TRANCOUNT is set to 0. Therefore, any further ROLLBACK TRAN or COMMIT TRAN instances will fail, so you do need to have error handling.

Try to avoid nesting transactions where possible, especially when one stored procedure calls another stored procedure within a transaction. It is not “wrong,” but it does require a great deal of care.

## Part 3: Deleting Data

Deleting data can be considered very straightforward, especially compared to all of the other data manipulation functions covered previously, particularly transactions and basic SQL. However, mistakes made when deleting data are very hard to recover from. Therefore, you must treat deleting data with the greatest of care and attention to detail, and especially test any joins and filtering via a SELECT statement before running the delete operation.

Deleting data without the use of a transaction is almost a final act: the only way to get the data back is to reenter it, restore it from a backup, or retrieve the data from any audit tables that had the data stored in them when the data was created. Deleting data is not like using the recycle bin on a Windows machine: unless the data is within a transaction, it is lost. Keep in mind that even if you use a transaction, the data will be lost once the transaction is committed. That's why it's very important to back up your database before running any major data modifications.

This section of the chapter will demonstrate the DELETE T-SQL syntax and then show how to use this within Query Editor. It is also possible to delete rows from the results pane within SQL Server, which will also be demonstrated.

However, what about when you want to remove all the rows within a table, especially when there could be thousands of rows to remove? You will find that the DELETE statement takes a very long time to run, as each row to delete is logged in the transaction log, thus allowing transactions to be rolled back. Luckily, there is a statement for this scenario, called TRUNCATE, which is covered in the section "Truncating a Table", introduced in a later section.

### **DELETE Syntax**

The DELETE statement is very short and sweet. To run the statement, simply state the table you wish to delete rows from, as shown here:

```
DELETE  
[FROM] tablename  
WHERE where_condition
```

The FROM is optional, so your syntax could easily read

```
DELETE tablename  
WHERE where_condition
```

There is nothing within this statement that has not been covered in other chapters. The only area that really needs to be mentioned is that rows can only be deleted from one table at a time, although when looking for rows to delete, you can join to several tables, as you can with SELECT and UPDATE.

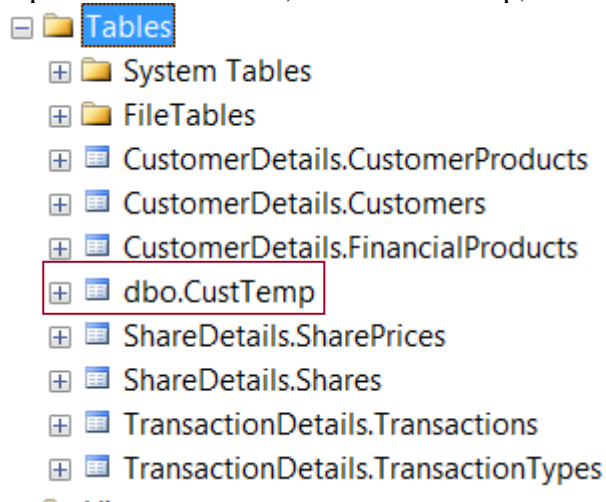
## **Practice: the DELETE Statement**

In this practice, we firstly created a table with the SELECT INTO statement called CustTemp. Rather than delete data from the main tables created so far, we'll use this temporary table in this section.

1. In an empty Query Editor window, enter the following code:

```
SELECT CustomerFirstName + ' ' + CustomerLastName AS [Name],  
ClearedBalance,UnclearedBalance  
INTO CustTemp  
FROM CustomerDetails.Customers
```

2. Execute the code. This will return the following message in the results pane:  
(3 row(s) affected)
3. If you now move to Object Explorer on the left-hand side (if Object Explorer is no longer there, press F8) and complete a refresh, you should see a new table in the expanded Tables node, called CustTemp, as shown in the following figure:



**Note:** You should use the INTO clause with care. For instance, in this example, security has not been set up for the table, and we are also creating tables within our database that have not been through any normalization or development life cycle. It is also very easy to fill up a database with these tables if we are not careful. However, it is a useful and handy method for taking a backup of a table and then working on that backup while testing out any queries that might modify the data. Do ensure though that there is enough space within the database before building the table if you do use this technique.

***It is best to avoid doing this in a production environment unless you really do need to keep the table permanently***

Now, we'll use transactions a great deal here to avoid having to keep inserting data back into the table. It's a good idea to use transactions for any type of table modification in your application. Imagine that you're at the ATM and you are transferring money from

your savings account to your checking account. During that process, a transaction built up of many actions is used to make sure that your money doesn't credit one system and not the other. If an error occurs, the entire transaction rolls back, and no money will move between the accounts.

Let's take a look at what happens if you were to run this statement:

```
BEGIN TRAN  
DELETE CustTemp
```

When this code runs, SQL Server opens a transaction and then tentatively deletes all the rows from the CustTemp table. The rows are not actually deleted until a COMMIT TRAN statement is issued. In the interim, though, SQL Server will place a lock on the rows of the table, or if this was a much larger table, SQL Server may decide that a table lock (locking the whole table to prevent other modifications) is better. Because of this lock, all users trying to modify data from this table will have to wait until a COMMIT TRAN or ROLLBACK TRAN statement has been issued and completed. If one is never issued, users will be blocked. This problem is one of a number of issues frequently encountered in applications when analyzing performance issues. Therefore, never have a BEGIN TRAN without a COMMIT TRAN or ROLLBACK TRAN.

1. Enter the following statements in an empty Query Editor pane. This will remove all the rows from our table within a transaction, prove the point by trying to list the rows, and then roll back the changes so that the rows are put back into the table.

```
BEGIN TRAN  
    SELECT * FROM CustTemp  
    DELETE CustTemp  
    SELECT * FROM CustTemp  
ROLLBACK TRAN  
SELECT * FROM CustTemp
```

2. Execute the code. You should see the results displayed in the following figure. Notice that the number of rows in the CustTemp table before the delete is 3, then after the delete the row count is tentatively set to 0. Finally, after the rollback, it's set back to 3. If we do not issue a ROLLBACK TRAN statement in here, we would see 0 rows, but other connections would be blocked until we did.



Results		Messages	
	Name	ClearedBala...	UnclearedBala...
1	Bernie McGlynn	4311.22	0.00
2	Julie Dewson	53.32	-12.21
3	Kirsty Hull	1266.00	10.32

	Name	ClearedBala...	UnclearedBala...
--	------	----------------	------------------

	Name	ClearedBala...	UnclearedBala...
1	Bernie McGlynn	4311.22	0.00
2	Julie Dewson	53.32	-12.21
3	Kirsty Hull	1266.00	10.32

3. Let's take this a stage further and only remove the last three rows of the table. Again, this will be within a transaction. Alter the preceding code as indicated in the following snippet. Here we are using the TOP statement to delete three random rows. Why random? SQL Server only stores rows in a definite order if they are covered by a clustered index. No other index, or no index, can guarantee the order in which SSE stores other rows. This is not the best way to delete rows, as in virtually all cases you will want to control the deletion.

```
BEGIN TRAN
    SELECT * FROM CustTemp
    DELETE TOP (2) CustTemp
    SELECT * FROM CustTemp
ROLLBACK TRAN
SELECT * FROM CustTemp
```

4. Execute the code, which should produce the results shown in the following figure.

Results		Messages	
	Name	ClearedBala...	UnclearedBala...
1	Bernie McGlynn	4311.22	0.00
2	Julie Dewson	53.32	-12.21
3	Kirsty Hull	1266.00	10.32

	Name	ClearedBala...	UnclearedBala...
1	Kirsty Hull	1266.00	10.32

	Name	ClearedBala...	UnclearedBala...
1	Bernie McGlynn	4311.22	0.00
2	Julie Dewson	53.32	-12.21
3	Kirsty Hull	1266.00	10.32

## Part 4: Truncating and Dropping a Table

### Truncating a Table

All delete actions caused by DELETE statements are recorded in the transaction log. Each time a row is deleted, a record is made of that fact. If you are deleting millions of rows before committing your transaction, your transaction log can grow quickly. Recall from earlier in the chapter the discussions about transactions; now think about this a bit more. What if the table you are deleting from has thousands of rows? That is a great deal of logging going on within the transaction log. But what if the deletion of these thousands of rows was, in fact, cleaning out all the data from the table to start afresh? Or perhaps this is some sort of transient table? Performing a DELETE would seem a lot of overhead when you don't really need to keep a log of the data deletions anyway. If the action failed for whatever reason, you would simply retry removing the rows a second time. This is where the TRUNCATE TABLE statement comes into its own.

By issuing a TRUNCATE TABLE statement, you are instructing SQL Server to delete every row within a table, without any logging or transaction processing taking place. In reality, minimal data is logged about what data pages have been deallocated and therefore removed from the database. This is in contrast to a DELETE statement, which will only deallocate and remove the pages from the table if it can get sufficient locks on the table to do this. The deletion of the rows can be almost instantaneous, and a great deal faster than

using the DELETE statement. This occurs not only because of the differences with what happens with the transaction log, but also because of how the data is locked at the time of deletion. Let's clarify this point before progressing.

When a DELETE statement is issued, each row that is to be deleted will be locked by SQL Server so that no modifications or other DELETE statements can attempt to work with that row. Deleting hundreds or thousands of rows is a large number of actions for SSE to perform, and it will take time to locate each row and place a lock against it. However, a TRUNCATE TABLE statement locks the whole table. This is one action that will prevent any data insertion, modification, or deletion from taking place.

The syntax for truncating a table is simple:

```
TRUNCATE TABLE [{database.schema_name.}] table
```

One "side effect" to the TRUNCATE TABLE clause is that it reseeds any identity columns. For example, say that you have a table with an identity column that is currently at 2,000,000. After truncating the table, the first inserted piece of data will produce the value 1 (if the seed is set to 1). If you issue a DELETE statement to delete the rows from the table, the first piece of data inserted after the table contents have been deleted will produce a value of 2,000,001, even though this newly inserted piece of data may be the only row in the table!

One of the limitations with the TRUNCATE TABLE statement is that you cannot issue it against tables that have foreign keys referencing them. For example, the CustomerDetails.Customers table has a foreign key referencing the TransactionDetails.Transactions table. If you try to issue the following statement:

```
TRUNCATE TABLE CustomerDetails.Customers
```

You will receive the following error message:

```
Msg 4712, Level 16, State 1, Line 1
```

```
Cannot truncate table 'CustomerDetails.Customers' because it is being  
referenced
```

```
by a FOREIGN KEY constraint.
```

### **Dropping a Table**

Another way to quickly delete the data in a table is to just delete the table and re-create it. Don't forget that if you do this, you will need to also re-create any constraints, indexes, and foreign keys. When you do this, SQL Server will deallocate the table, which is minimally logged. To drop a table in SQL Server, issue the following statement:

```
DROP TABLE table_name
```

As with TRUNCATE TABLE, DROP TABLE cannot be issued against a table that has a foreign key referencing it. In this situation, either the foreign key constraint referencing the table or the referencing table itself must first be dropped before it is possible to drop the original table.