

CHAPTER 2



Perform, Capture, and Track Data Modifications

In this chapter, I review how to modify data using the Transact-SQL INSERT, UPDATE, and DELETE commands. I'll review the basics of each command and cover specific techniques such as inserting data from a stored procedure and importing an image file into a table using OPENROWSET BULK functionality.

The new SQL Server 2008 features I cover in this chapter include the following:

- Inserting multiple rows from a single INSERT statement. I'll also demonstrate using the multiple-row technique to create a query data source in a SELECT clause (without having to create a permanent or temporary table).
- New assignment operators that allow you to modify a passed data value with minimal coding.
- The new MERGE command, which allows you to consolidate and apply data modification commands using a single block of code.
- Storing unstructured data on the file system while maintaining SQL Server transactional control using the new FILESTREAM attribute.
- Two new options for tracking table data changes using Change Data Capture (CDC) and Change Tracking built-in functionality.

Before going into the new features, however, I'll start the chapter off by reviewing basic INSERT concepts.

INSERT

The simplified syntax for the INSERT command is as follows:

```
INSERT
[ INTO]
table_or_view_name
[ ( column_list ) ]
VALUES (( {DEFAULT | NULL | expression } [ ,...n ] ) [ ,...n ])
```

The arguments of this command are described in Table 2-1.

Table 2-1. *INSERT Command Arguments*

Argument	Description
table_or_view_name	The name of the table or updateable view that you are inserting a row into.
column_list	The explicit comma-separated list of columns on the insert table that will be populated with values.
(DEFAULT NULL expression){ ,...n]	The comma-separated list of values to be inserted as a row into the table. In SQL Server 2008, you can insert multiple rows in a single statement. Each value can be an expression, NULL value, or DEFAULT value (if a default was defined for the column).

Inserting a Row into a Table

In this recipe, I demonstrate the use of INSERT to add new rows into a table (as specified by table_name), specifying a column_list of columns into which the data should be inserted, and a corresponding comma-separated list of values to be inserted, [, ...n], in the VALUES clause. Specifically, here I demonstrate inserting a single row into the AdventureWorks Production.Location table:

```
USE AdventureWorks
GO
```

```
INSERT Production.Location
(Name, CostRate, Availability)
VALUES ('Wheel Storage', 11.25, 80.00)
```

This returns

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

This next query then searches for any row with the name Wheel Storage:

```
SELECT Name,
       CostRate,
       Availability
FROM Production.Location
WHERE Name = 'Wheel Storage'
```

This returns

Name	CostRate	Availability
Wheel Storage	11.25	80.00

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

How It Works

In this recipe, a new row was inserted into the Production.Location table.

The query began with the INSERT command and the name of the table that will receive the inserted data (the INTO keyword is optional):

```
INSERT Production.Location
```

The next line of code explicitly listed the columns of the destination table that I wish to insert the data into:

```
(Name, CostRate, Availability)
```

A comma must separate each column. Columns don't need to be listed in the same order as they appear in the base table—as long as the values in the VALUES clause exactly match the order of the column list. Column lists are not necessary if the values are all provided and are in the same order. However, using column lists should be required for your production code, particularly if the base schema undergoes periodic changes. This is because explicitly listing columns allows you to add new columns to the base table without changing the referencing code (assuming the new column has a default value).

The next line of code was the VALUES clause and a comma-separated list of values (expressions) to insert:

```
VALUES ('Wheel Storage', 11.25, 80.00)
```

As I've noted previously, the values in this list must be provided in the same order as the listed columns or, if no columns are listed, the same order of the columns in the table.

Inserting a Row Using Default Values

In this recipe, I'll show you how to load a row into a table such that it takes a default value for a certain column (or columns), using the DEFAULT keyword. In the previous recipe, the `Production.Location` table had a row inserted into it. The `Production.Location` table has two other columns that were not explicitly referenced in the INSERT statement. If you look at the column definition of Table 2-2, you'll see that there is also a `LocationID` and a `ModifiedDate` column that were not included in the previous example's INSERT.

Table 2-2. *Production.Location Table Definition*

Column Name	Data Type	Nullability	Default Value	Identity Column?
LocationID	smallint	NOT NULL		Yes
Name	dbo.Name (user-defined data type)	NOT NULL		No
CostRate	smallmoney	NOT NULL	0.00	No
Availability	decimal(8,2)	NOT NULL	0.00	No
ModifiedDate	datetime	NOT NULL	GETDATE() (function to return the current date and time)	No

Note See Chapter 4 for more information on the CREATE TABLE command, IDENTITY columns, and DEFAULT values.

The `ModifiedDate` column has a default value that populates the current date and time for new rows if the column value wasn't explicitly inserted. The INSERT could have been written to update this column too. For example:

```
INSERT Production.Location
(Name, CostRate, Availability, ModifiedDate)
VALUES ('Wheel Storage 2', 11.25, 80.00, '1/1/2005')
```

When a column has a default value specified in a table, you can use the `DEFAULT` keyword in the `VALUES` clause, in order to explicitly trigger the default value.

For example:

```
INSERT Production.Location
(Name, CostRate, Availability, ModifiedDate)
VALUES ('Wheel Storage 3', 11.25, 80.00, DEFAULT)
```

If each column in the table uses defaults for all columns, you can trigger an insert that inserts a row using only the defaults by including the `DEFAULT VALUES` option. For example:

```
INSERT dbo.ExampleTable
DEFAULT VALUES
```

How It Works

The `DEFAULT` keyword allows you to explicitly set a column's default value in an `INSERT` statement. The `DEFAULT VALUES` keywords can be used in your `INSERT` statement to explicitly set all the column's default values (assuming the table is defined with a default on each column).

The `LocationID` column from the `Production.Location` table, however, is an `IDENTITY` column (not a defaulted column). An `IDENTITY` property on a column causes the value in that column to automatically populate with an incrementing numeric value. Because `LocationID` is an `IDENTITY` column, the database manages inserting the values for this row, so an `INSERT` statement cannot normally specify a value for an `IDENTITY` column. If you want to specify a certain value for an `IDENTITY` column, you need to follow the procedure outlined in the next recipe.

Explicitly Inserting a Value into an IDENTITY Column

In this recipe, I'll demonstrate how to explicitly insert values into an `IDENTITY` property column. A column using an `IDENTITY` property automatically increments based on a numeric seed value and incrementing value for every row inserted into the table. `IDENTITY` columns are often used as surrogate keys (a *surrogate key* is a unique primary key generated by the database that holds no business-level significance other than to ensure uniqueness within the table).

In data load or recovery scenarios, you may find that you need to manually insert explicit values into an `IDENTITY` column. For example, if a row with the key value of 4 were deleted accidentally, and you needed to manually reconstruct that row, preserving the original value of 4 with the old business information, you would need to be able to explicitly insert this value into the table.

To explicitly insert a numeric value into a column using an `IDENTITY` property, you must use the `SET IDENTITY_INSERT` command. The syntax is as follows:

```
SET IDENTITY_INSERT [ database_name . [ schema_name ] . ] table { ON | OFF }
```

The arguments of this command are described in Table 2-3.

Table 2-3. *SET IDENTITY_INSERT Command*

Argument	Description
[database_name . [schema_name] .]	These specify the optional database name, optional schema name, and required table name for which explicit values will be allowed to be inserted into table an <code>IDENTITY</code> property column.
ON OFF	When set ON, explicit value inserts are allowed. When OFF, explicit value inserts are <i>not</i> allowed.

In this recipe, I'll demonstrate how to explicitly insert the value of an IDENTITY column into a table. The following query first demonstrates what happens if you try to do an explicit insert into an identity column without first using IDENTITY_INSERT:

```
INSERT HumanResources.Department
(DepartmentID, Name, GroupName)
VALUES (17, 'Database Services', 'Information Technology')
```

This returns an error, keeping you from inserting an explicit value for the identity column:

```
Msg 544, Level 16, State 1, Line 2
Cannot insert explicit value for identity column in table 'Department' when
IDENTITY_INSERT is set to OFF.
```

Using SET IDENTITY_INSERT removes this barrier, as this next example demonstrates:

```
SET IDENTITY_INSERT HumanResources.Department ON

INSERT HumanResources.Department
(DepartmentID, Name, GroupName)
VALUES (17, 'Database Services', 'Information Technology')

SET IDENTITY_INSERT HumanResources.Department OFF
```

How It Works

In the recipe, this property was set ON prior to the insert:

```
SET IDENTITY_INSERT HumanResources.Department ON
```

The INSERT was then performed using a value of 17. When inserting into an identity column, you must also explicitly list the column names after the INSERT table_name clause:

```
INSERT HumanResources.D epartment
(DepartmentID, Name, GroupName)
VALUES (17, 'Database Services', 'Information Technology')
```

For inserted values greater than the current identity value, new inserts to the table will automatically use the new value as the identity seed.

IDENTITY_INSERT should be set OFF once you are finished explicitly inserting values:

```
SET IDENTITY_INSERT HumanResources.Department OFF
```

You should set this OFF once you are finished, as only one table in the session (your database connection session) can have IDENTITY_INSERT ON at the same time (assuming that you wish to insert explicit values for multiple tables). Closing your session will remove the ON property, setting it back to OFF.

Inserting a Row into a Table with a uniqueidentifier Column

In this recipe, I'll show you how to insert data into a table that uses a uniqueidentifier column. This data type is useful in scenarios where your identifier *must* be unique across several SQL Server instances. For example, if you have ten remote SQL Server instances generating records that are then consolidated on a single SQL Server instance, using an IDENTITY value generates the risk of primary key conflicts. Using a uniqueidentifier data type would allow you to avoid this.

A `uniqueidentifier` data type stores a 16-byte globally unique identifier (GUID) that is often used to ensure uniqueness across tables within the same or a different database. GUIDs offer an alternative to integer value keys, although their width compared to integer values can sometimes result in slower query performance for bigger tables.

To generate this value for a new `INSERT`, the `NEWID` system function is used. `NEWID` generates a unique `uniqueidentifier` data type value, as this recipe demonstrates:

```
INSERT Purchasing.ShipMethod
(Name, ShipBase, ShipRate, rowguid)
VALUES('MIDDLETON CARGO TS1', 8.99, 1.22, NEWID())
```

```
SELECT rowguid, name
FROM Purchasing.ShipMethod
WHERE Name = 'MIDDLETON CARGO TS1'
```

This returns the following (if you are following along, note that your `Rowguid` value will be different from mine):

Rowguid	name
174BE850-FDEA-4E64-8D17-C019521C6C07	MIDDLETON CARGO TS1

How It Works

The `rowguid` column in the `Purchasing.ShipMethod` table is a `uniqueidentifier` data type column. Here is an excerpt from the table definition:

```
rowguid uniqueidentifier ROWGUIDCOL NOT NULL DEFAULT (newid()),
```

To generate a new `uniqueidentifier` data type value for this inserted row, the `NEWID()` function was used in the `VALUES` clause:

```
VALUES('MIDDLETON CARGO TS1', 8.9 9, 1.2 2, NEWID())
```

Selecting the new row that was just created, the `rowguid` was given a `uniqueidentifier` value of `174BE850-FDEA-4E64-8D17-C019521C6C07` (although when you test it yourself, you'll get a different value because `NEWID` creates a new value each time it is executed).

Inserting Rows Using an `INSERT...SELECT` Statement

The previous recipes showed you how to insert a single row of data. In this recipe, I'll show you how to insert multiple rows into a table using `INSERT...SELECT`. The syntax for performing an `INSERT...SELECT` operation is as follows:

```
INSERT
[ INTO]
table_or_view_name
[ ( column_list ) ]
SELECT column_list FROM data_source
```

The syntax for using `INSERT...SELECT` is almost identical to inserting a single row, only instead of using the `VALUES` clause, you designate a `SELECT` query that will populate the columns and rows into the table or updateable view. The `SELECT` query can be based on one or more data sources, so long as the column list conforms to the expected data types of the destination table.

For the purposes of this example, a new table will be created for storing the rows. The example populates values from the `HumanResources.Shift` table into the new `dbo.Shift_Archive` table:

```

CREATE TABLE [dbo].[Shift_Archive](
    [ShiftID] [tinyint] NOT NULL,
    [Name] [dbo].[Name] NOT NULL,
    [StartTime] [datetime] NOT NULL,
    [EndTime] [datetime] NOT NULL,
    [ModifiedDate] [datetime] NOT NULL DEFAULT (getdate()),
    CONSTRAINT [PK_Shift_ShiftID] PRIMARY KEY CLUSTERED
    ([ShiftID] ASC)
)
GO

```

Next, an INSERT..SELECT is performed:

```

INSERT dbo.Shift_Archive
(ShiftID, Name, StartTime, EndTime, ModifiedDate)
SELECT ShiftID, Name, StartTime, EndTime, ModifiedDate
FROM HumanResources.Shift
ORDER BY ShiftID

```

The results show that three rows were inserted:

(3 row(s) affected)

Next, a query is executed to confirm the inserted rows in the Shift_Archive table:

```

SELECT ShiftID, Name
FROM Shift_Archive

```

This returns

ShiftID	Name
1	Day
2	Evening
3	Night

(3 row(s) affected)

How It Works

Using the INSERT...SELECT statement, you can insert multiple rows into a table based on a SELECT query. Just like regular, single-value INSERTs, you begin by using INSERT table_name and the list of columns to be inserted into the table (in parentheses):

```

INSERT Shift_Archive
(ShiftID, Name, StartTime, EndTime, ModifiedDate)

```

Following this is the query used to populate the table. The SELECT statement must return columns in the same order as the columns appear in the INSERT column list. The columns list must also have data type compatibility with the associated INSERT column list:

```

SELECT ShiftID, Name, StartTime, EndTime, ModifiedDate
FROM HumanResources.Shift
ORDER BY ShiftID

```

When the column lists aren't designated, the SELECT statement must provide values for *all* the columns of the table into which the data is being inserted.

Inserting Data from a Stored Procedure Call

In this recipe, I demonstrate how to insert table data by using a stored procedure. A *stored procedure* groups one or more Transact-SQL statements into a logical unit and stores it as an object in a SQL Server database. Stored procedures allow for more sophisticated result set creation (for example, you can use several intermediate result sets built in temporary tables before returning the final result set). Reporting system stored procedures that return a result set can also be used for `INSERT...EXEC`, which is useful if you wish to retain SQL Server information in tables.

This recipe also teaches you how to add rows to a table based on the output of a stored procedure. A stored procedure can only be used in this manner if it returns data via a `SELECT` command from within the procedure definition and the result set (or even multiple result sets) match the required number of supplied values to the `INSERT`.

Note For more information on stored procedures, see Chapter 10.

The syntax for inserting data from a stored procedure is as follows:

```
INSERT
  [ INTO]
  table_or_view_name
  [ ( column_list ) ]
  EXEC stored_procedure_name
```

The syntax is almost identical to the previously demonstrated `INSERT` examples, only this time the data is populated via an executed stored procedure.

In this example, a stored procedure is created that returns rows from the `Production.TransactionHistory` table based on the begin and end dates passed to the stored procedure. These results returned by the procedure also only return rows that don't exist in the `Production.TransactionHistoryArchive`:

```
CREATE PROCEDURE dbo.usp_SEL_Production_TransactionHistory
  @ModifiedStartDT datetime,
  @ModifiedEndDT datetime
AS

SELECT TransactionID, ProductID, ReferenceOrderID, ReferenceOrderLineID,
  TransactionDate, TransactionType, Quantity, ActualCost, ModifiedDate
FROM Production.TransactionHistory
WHERE ModifiedDate BETWEEN @ModifiedStartDT AND @ModifiedEndDT AND
  TransactionID NOT IN
  (SELECT TransactionID
   FROM Production.TransactionHistoryArchive)

GO
```

Next, this example tests the stored procedures to precheck which rows will be inserted:

```
EXEC dbo.usp_SEL_Production_TransactionHistory '6/2/04', '6/3/04'
```

This returns 568 rows based on the date range passed to the procedure. In the next example, this stored procedure is used to insert the 568 rows into the `Production.TransactionHistoryArchive` table:

```

INSERT Production.TransactionHistoryArchive
(TransactionID, ProductID, ReferenceOrderID, ReferenceOrderLineID, TransactionDate,
TransactionType, Quantity, ActualCost, ModifiedDate)
EXEC dbo.usp_SEL_Production_TransactionHistory '6/2/04', '6/3/04'

```

How It Works

This example demonstrated using a stored procedure to populate a table using INSERT and EXEC. The INSERT began with the name of the table to be inserted into:

```
INSERT Production.TransactionHistoryArchive
```

Next was the list of columns to be inserted into:

```
(TransactionID, ProductID, ReferenceOrderID, ReferenceOrderLineID,
TransactionDate, TransactionType, Quantity, ActualCost, ModifiedDate)
```

Last was the EXEC statement, which executed the stored procedures. Any parameters the stored procedure expects follow the stored procedure name:

```
EXEC usp_SEL_Production_TransactionHistory '6/2/04', '6/3/04'
```

Inserting Multiple Rows with VALUES

SQL Server 2008 introduces the ability to insert multiple rows using a single INSERT command without having to issue a subquery or stored procedure call. This allows the application to reduce the code required to add multiple rows and also reduce the number of individual commands executed. Essentially, you use the VALUES to group and specify one or more rows and their associated column values, as the following recipe demonstrates:

```

-- Create a lookup table
CREATE TABLE HumanResources.Degree
(DegreeID int NOT NULL IDENTITY(1,1) PRIMARY KEY,
DegreeNM varchar(30) NOT NULL,
DegreeCD varchar(5) NOT NULL,
ModifiedDate datetime NOT NULL)
GO

INSERT HumanResources.Degree
(DegreeNM, DegreeCD, ModifiedDate)
VALUES
('Bachelor of Arts', 'B.A.', GETDATE()),
('Bachelor of Science', 'B.S.', GETDATE()),
('Master of Arts', 'M.A.', GETDATE()),
('Master of Science', 'M.S.', GETDATE()),
('Associate's Degree', 'A.A.', GETDATE())
GO

```

This returns the following query output:

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

How It Works

In this recipe, I demonstrated inserting multiple rows from a single INSERT statement. I started off by creating a new table to hold information on college degree types. I then used the INSERT in a typical fashion, showing the column names that would have values passed to it for each row:

```
INSERT HumanResources.Degree
(DegreeNM, DegreeCD, ModifiedDate)
```

Next, in the VALUES clause, I designated a new row for each degree type. Each row had three columns, and these columns were encapsulated in parentheses:

```
VALUES
('Bachelor of Arts', 'B.A.', GETDATE()),
('Bachelor of Science', 'B.S.', GETDATE()),
('Master of Arts', 'M.A.', GETDATE()),
('Master of Science', 'M.S.', GETDATE()),
('Associate's Degree', 'A.A.', GETDATE())
GO
```

This new feature allowed me to insert multiple rows without having to retype the initial INSERT table name and column list. An example of where this may be useful would be for custom applications that include a database schema along with a set of associated lookup values. Rather than hand-code 50 INSERT statements in your setup script, you can create a single INSERT with multiple rows designated. This also allows you to bypass importing a rowset from an external source.

Using VALUES As a Table Source

The previous recipe demonstrated how to insert multiple rows without having to retype the initial INSERT table name and column list. Using this same new feature in SQL Server 2008, you can also reference the VALUES list in the FROM clause of a SELECT statement.

This recipe will demonstrate how to reference a result set without having to use a permanent or temporary table. The following query demonstrates listing various college degrees in a five-row result set—without having to persist the rows in a table or reference in a subquery:

```
SELECT DegreeNM, DegreeCD, ModifiedDT
FROM
(VALUE
('Bachelor of Arts', 'B.A.', GETDATE()),
('Bachelor of Science', 'B.S.', GETDATE()),
('Master of Arts', 'M.A.', GETDATE()),
('Master of Science', 'M.S.', GETDATE()),
('Associate's Degree', 'A.A.', GETDATE()))
Degree (DegreeNM, DegreeCD, ModifiedDT)
```

This returns

DegreeNM	DegreeCD	ModifiedDT
Bachelor of Arts	B.A.	2007-08-21 19:10:34.667
Bachelor of Science	B.S.	2007-08-21 19:10:34.667
Master of Arts	M.A.	2007-08-21 19:10:34.667
Master of Science	M.S.	2007-08-21 19:10:34.667
Associate's Degree	A.A.	2007-08-21 19:10:34.667

(5 row(s) affected)

How It Works

This recipe demonstrated using a new SQL Server 2008 technique for returning a result set to persist the rows in storage. Breaking down the query, the first row in the SELECT clause listed the column names:

```
SELECT DegreeNM, DegreeCD, ModifiedDT
```

These are not actual column names from a referenced table—but instead are aliased names I defined later on in the query itself.

The next line defined the FROM clause for the data source, followed by a parenthesis encapsulating the VALUES keyword:

```
FROM  
(VALUES
```

The next few lines of code listed rows I wished to return from this query (similar to how I inserted multiple rows in a single INSERT in the previous recipe):

```
('Bachelor of Arts', 'B.A.', GETDATE()),  
( 'Bachelor of Science', 'B.S.', GETDATE()),  
( 'Master of Arts', 'M.A.', GETDATE()),  
( 'Master of Science', 'M.S.', GETDATE()),  
( 'Associate's Degree', 'A.A.', GETDATE())  
)
```

Lastly, after the final closing parenthesis for the row list, I defined a name for this data source and the associated column names to be returned for each column (and to be referenced in the SELECT clause):

```
Degree (DegreeNM, DegreeCD, ModifiedDT)
```

This new technique allowed me to specify rows of a table source without having to actually create a temporary or permanent table.

UPDATE

The following is basic syntax for the UPDATE statement:

```
UPDATE <table_or_view_name>  
SET column_name = {expression | DEFAULT | NULL} [ ,...n ]  
WHERE <search_condition>
```

The arguments of this command are described in Table 2-4.

Table 2-4. UPDATE Command Arguments

Argument	Description
table_or_view_name	The table or updateable view containing data to be updated.
column_name = {expression DEFAULT NULL}	The name of the column or columns to be updated. The column can be set to an expression, the DEFAULT value of the column, or a NULL.
search_condition	The search condition that defines <i>what</i> rows are modified. If this isn't included, all rows from the table or updateable view will be modified.

Updating a Single Row

In this recipe, I'll demonstrate how to use the UPDATE statement to modify data. With the UPDATE statement, you can apply changes to single or multiple columns, as well as to single or multiple rows.

In this example, a single row is updated by designating the SpecialOfferID, which is the primary key of the table (for more on primary keys, see Chapter 4). Before performing the update, I'll first query the specific row that I plan on modifying:

```
SELECT DiscountPct
FROM Sales.SpecialOffer
WHERE SpecialOfferID = 10
```

This returns

```
DiscountPct
0.50
```

Now I'll perform the modification:

```
UPDATE Sales.SpecialOffer
SET DiscountPct = 0.15
WHERE SpecialOfferID = 10
```

Querying that specific row after the update confirms that the value of DiscountPct was indeed modified:

```
SELECT DiscountPct
FROM Sales.SpecialOffer
WHERE SpecialOfferID = 10
```

This returns

```
DiscountPct
0.15
```

How It Works

In this example, the query started off with UPDATE and the table name Sales.SpecialOffer:

```
UPDATE Sales.SpecialOffer
```

Next, SET was used, followed by the column name to be modified, and an equality operator to modify the DiscountPct to a value of 0.15. Relating back to the syntax at the beginning of the recipe, this example is setting the column to an expression value, and not a DEFAULT or NULL value:

```
SET DiscountPct = 0.15
```

Had this been the end of the query, *all* rows in the Sales.SpecialOffer table would have been modified, because the UPDATE clause works at the table level, not the row level. But the intention of this query was to only update the discount percentage for a specific product. The WHERE clause was used in order to achieve this:

```
WHERE SpecialOfferID = 10
```

After executing this query, only one row is modified. Had there been multiple rows that met the search condition in the WHERE clause, those rows would have been modified too.

Tip Performing a SELECT query with the FROM and WHERE clauses of an UPDATE, prior to the UPDATE, allows you to see what rows you will be updating (an extra validation that you are updating the proper rows). This is also a good opportunity to use a transaction to allow for rollbacks in the event that your modifications are undesired. For more on transactions, see Chapter 3.

Updating Rows Based on a FROM and WHERE Clause

In this recipe, I'll show you how to use the UPDATE statement to modify rows based on a FROM clause and associated WHERE clause search conditions. The basic syntax, elaborating from the last example, is as follows:

```
UPDATE <table_or_view_name>
SET column_name = {expression | DEFAULT | NULL} [ ,...n ]
FROM <table_source>
WHERE <search_condition>
```

The FROM and WHERE clauses are not mandatory; however, you will find that they are almost always implemented in order to specify exactly which rows are to be modified, based on joins against one or more tables.

In this example, assume that a specific product, “Full-Finger Gloves, M,” from the Production.Product table has a customer purchase limit of two units per customer. For this query's requirement, any shopping cart with a quantity of more than two units for this product should immediately be adjusted back to the required limit:

```
UPDATE Sales.ShoppingCartItem
SET Quantity =2,
    ModifiedDate = GETDATE()
FROM Sales.ShoppingCartItem c
INNER JOIN Production.Product p ON
    c.ProductID = p.ProductID
WHERE p.Name = 'Full-Finger Gloves, M ' AND
    c.Quantity > 2
```

How It Works

Stepping through the code, the first line showed the table to be updated:

```
UPDATE Sales.ShoppingCartItem
```

Next, the columns to be updated were designated in the SET clause:

```
SET Quantity =2,
    ModifiedDate = GETDATE()
```

Next came the optional FROM clause where the Sales.ShoppingCartItem and Production.Product tables were joined by ProductID. As you can see, the object being updated can also be referenced in the FROM clause. The reference in the UPDATE and in the FROM were treated as the same table:

```
FROM Sales.ShoppingCartItem c
INNER JOIN Production.Product p ON
    c.ProductID = p.ProductID
```

Using the updated table in the FROM clause allows you to join to other tables. Presumably, those other joined tables will be used to filter the updated rows or to provide values for the updated rows.

If you are self-joining to more than one reference of the updated table in the FROM clause, at least one reference to the object *cannot* specify a table alias. All the other object references, however, would have to use an alias.

The WHERE clause specified that only the “Full-Finger Gloves, M” product in the Sales.ShoppingCartItem should be modified, and only if the Quantity is greater than 2 units:

```
WHERE p.Name = 'Full-Finger Gloves, M ' AND
      c.Quantity > 2
```

Updating Large Value Data Type Columns

In this recipe, I'll show you how to modify large-value data type column values. SQL Server introduced new large-value data types in the previous version, which were intended to replace the deprecated text, ntext, and image data types. These data types include

- varchar(max), which holds non-Unicode variable-length data
- nvarchar(max), which holds Unicode variable-length data
- varbinary(max), which holds variable-length binary data

These data types can store up to $2^{31}-1$ bytes of data (for more information on data types, see Chapter 4).

One of the major drawbacks of the old text and image data types is that they required you to use separate functions such as WRITETEXT and UPDATETEXT in order to manipulate the image/text data. Using the new large-value data types, you can now use regular INSERT and UPDATEs instead.

The syntax for inserting a large-value data type is no different from a regular insert. For updating large-value data types, however, the UPDATE command now includes the .WRITE method:

```
UPDATE <table_or_view_name>
SET column_name = .WRITE ( expression , @Offset , @Length )
FROM <table_source>
WHERE <search_condition>
```

The parameters of the .WRITE method are described in Table 2-5.

Table 2-5. UPDATE Command with .WRITE Clause

Argument	Description
expression	The expression defines the chunk of text to be placed in the column.
@Offset	@Offset determines the starting position in the existing data the new text should be placed. If @Offset is NULL, it means the new expression will be appended to the end of the column (also ignoring the second @Length parameter).
@Length	@Length determines the length of the section to overlay.

This example starts off by creating a table called RecipeChapter:

```
CREATE TABLE dbo.RecipeChapter
(ChapterID int NOT NULL,
 Chapter varchar(max) NOT NULL)
GO
```

Next, a row is inserted into the table. Notice that there is nothing special about the string being inserted into the Chapter varchar(max) column:

```
INSERT dbo.RecipeChapter
(ChapterID, Chapter)
VALUES
(1, 'At the beginning of each chapter you will notice
that basic concepts are covered first.' )
```

This next example updates the newly inserted row, adding a sentence to the end of the existing sentence:

```
UPDATE RecipeChapter
SET Chapter .WRITE (' In addition to the basics, this chapter will also provide
recipes that can be used in your day to day development and administration.' ,
NULL, NULL)
WHERE ChapterID = 1
```

Next, for that same row, the phrase “day to day” is replaced with the single word “daily”:

```
UPDATE RecipeChapter
SET Chapter .WRITE('daily', 181, 10)
WHERE ChapterID = 1
```

Lastly, the results are returned for that row:

```
SELECT Chapter
FROM RecipeChapter
WHERE ChapterID = 1
```

This returns

```
Chapter
At the beginning of each chapter you will notice that basic concepts
are covered first.
In addition to the basics, this chapter will also provide recipes
that can be used in your daily development and administration.
```

How It Works

The recipe began by creating a table where book chapter descriptions would be held. The Chapter column used a varchar(max) data type:

```
CREATE TABLE RecipeChapter
(ChapterID int NOT NULL,
Chapter varchar(max) NOT NULL)
```

Next, a new row was inserted. Notice that the syntax for inserting a large-object data type doesn't differ from inserting data into a regular non-large-value data type:

```
INSERT RecipeChapter
(ChapterID, Chapter)
VALUES
(1, 'At the beginning of each chapter you will
notice that basic concepts are covered first.' )
```

Next, an UPDATE was performed against the RecipeChapter table to add a second sentence after the end of the first sentence:

```
UPDATE RecipeChapter
```

The SET command was followed by the name of the column to be updated (Chapter) and the new .WRITE command. The .WRITE command was followed by an open parenthesis, a single quote, and the sentence to be appended to the end of the column:

```
SET Chapter .WRITE(' In addition to the basics,
this chapter will also provide recipes that can be
used in your day to day development and administration.' ,
NULL, NULL)
```

The WHERE clause specified that the Chapter column for a single row matching ChapterID = 1 be modified:

```
WHERE ChapterID = 1
```

The next example of .WRITE demonstrated replacing data within the body of the column. In the example, the expression day to day was replaced with daily. The bigint value of @Offset and @Length are measured in bytes for varbinary(max) and varchar(max) data types. For nvarchar(max), these parameters measure the actual number of characters. For the example, the .WRITE had a value for @Offset (181 bytes into the text) and @Length (10 bytes long):

```
UPDATE RecipeChapter
SET Chapter .WRITE('daily', 181, 10)
WHERE ChapterID = 1
```

Inserting or Updating an Image File Using OPENROWSET and BULK

In this recipe, I demonstrate how to insert or update an image file from the file system into a SQL Server table. Adding images to a table in earlier versions of SQL Server usually required the use of external application tools or scripts. There was no elegant way to insert images using just Transact-SQL.

As of SQL Server 2005 and 2008, UPDATE and OPENROWSET can be used together to import an image into a table. OPENROWSET can be used to import a file into a single-row, single-column value. The basic syntax for OPENROWSET as it applies to this recipe is as follows:

```
OPENROWSET
( BULK 'data_file' ,
  SINGLE_BLOB | SINGLE_CLOB | SINGLE_NCLOB )
```

The parameters for this command are described in Table 2-6.

Table 2-6. *The OPENROWSET Command Arguments*

Parameter	Description
data_file	This specifies the name and path of the file to read.
SINGLE_BLOB SINGLE_CLOB SINGLE_NCLOB	Designate the SINGLE_BLOB object for importing into a varbinary(max) data type. Designate SINGLE_CLOB for ASCII data into a varchar(max) data type and SINGLE_NCLOB for importing into a nvarchar(max) Unicode data type.

Note See Chapter 27 for a detailed review of the syntax of OPENROWSET.

The first part of the recipe creates a new table that will be used to store image files:

```
CREATE TABLE dbo.StockBmps
  (StockBmpID int NOT NULL,
   bmp varbinary(max) NOT NULL)
GO
```

Next, a row containing the image file will be inserted into the table:

```
INSERT dbo.StockBmps
  (StockBmpID, bmp)
SELECT 1,
  BulkColumn
FROM OPENROWSET(BULK
  'C:\Apress\StockPhotoOne.bmp', SINGLE_BLOB) AS x
```

This next query selects the row from the table:

```
SELECT bmp
FROM StockBmps
WHERE StockBmpID = 1
```

This returns the following (abridged) results:

```
bmp
0x424D365600000000000036040000280000007D000000A40000000100080000000000052000000000
000000000000010000000100001B71900057575E00EFEFEF000F0B0C0023A7D30028D2FF001A5B7
1005473A1008C8C8C00B3B3B300208BB00031303100D1D1D1005896B20018425600112C3500777D
7B00474F9100A089660078CDD0071AFC6009D9D9D0045444A00686B6F00728FAD0077998C001
C1D1E0009040500080304000501000026C4FF
```

The last example in this recipe updates an existing BMP file, changing it to a different BMP file:

```
UPDATE dbo.StockBmps
SET bmp =
  (SELECT BulkColumn
  FROM OPENROWSET(BULK 'C:\Apress\StockPhotoTwo.bmp', SINGLE_BLOB) AS x)
WHERE StockBmpID =1
```

How It Works

In this recipe, I've demonstrated using OPENROWSET with the BULK option to insert a row containing a BMP image file, and then the way to update it to a different GIF file.

First, a table was created to hold the GIF files using a varbinary(max) data type:

```
CREATE TABLE dbo.StockBmps
  (StockBmpID int NOT NULL,
   bmp varbinary(max) NOT NULL)
```

Next, a new row was inserted using INSERT:

```
INSERT dbo.StockBmps
  (StockBmpID, bmp)
```

The INSERT was populated using a SELECT query against the OPENROWSET function to bring in the file data. The BulkColumn referenced in the query represents the varbinary value to be inserted into the varbinary(max) row from the OPENROWSET data source:

```
SELECT 1,
  BulkColumn
```

In the FROM clause, OPENROWSET was called. OPENROWSET allows you to access remote data from a data source:

```
FROM OPENROWSET(BULK
'C:\Apress\StockPhotoOne.bmp', SINGLE_BLOB) AS x
```

The BULK option was used inside the function, followed by the file name and the SINGLE_BLOB keyword. The BULK option within OPENROWSET means that data will be read from a file (in this case, the BMP file specified after BULK). The SINGLE_BLOB switch tells OPENROWSET to return the contents of the data file as a single-row, single-column varbinary(max) rowset.

This recipe also demonstrates an UPDATE of the varbinary(max) column from an external file. The UPDATE designated the StockBmps table and used SET to update the bmp column:

```
UPDATE StockBmps
SET bmp =
```

The expression to set the new image to StockPhotoTwo.bmp from the previous StockPhotoOne.bmp occurred in a subquery. It used almost the same syntax as the previous INSERT; only this time the only value returned in the SELECT is the BulkColumn column:

```
(SELECT BulkColumn
FROM OPENROWSET(BULK 'C:\Apress\StockPhotoTwo.bmp', SINGLE_BLOB) AS x)
```

The image file on the machine was then stored in the column value for that row as varbinary(max) data.

Storing Unstructured Data on the File System While Maintaining SQL Server Transactional Control

SQL Server 2008 introduces the new FILESTREAM attribute, which can be applied to the varbinary(max) data type. Using FILESTREAM, you can exceed the 2GB limit on stored values and take advantage of relational handling of files via SQL Server, while actually storing the files on the file system. BACKUP and RESTORE operations maintain both the database data as well as the files saved on the file system, thus handling end-to-end data recoverability for applications that store both structured and unstructured data. FILESTREAM marries the transactional consistency capabilities of SQL Server with the performance advantages of NT file system streaming.

T-SQL is used to define the FILESTREAM attribute and can be used to handle the data; however, Win32 streaming APIs are the preferred method from the application perspective when performing actual read and write operations (specifically using the OpenSqlFilestream API). Although demonstrating Win32 and the implementation of applicable APIs is outside of the scope of this book, I will use this recipe to walk you through how to set up a database and table with the FILESTREAM attribute, INSERT a new row, and use a query to pull path and transaction token information that is necessary for the OpenSqlFilestream API call.

Tip FILESTREAM must be configured at both the Windows and SQL Server scope. To enable FILESTREAM for the Windows scope and define the associated file share, use SQL Server Configuration Manager. To enable FILESTREAM at the SQL Server instance level, use sp_configure with the filestream_access_level option.

To confirm whether FILESTREAM is configured for the SQL Server instance, I can validate the setting using the SERVERPROPERTY function and three different properties that describe the file share name of the filestream share and the associated effective and actual configured values:

```
SELECT SERVERPROPERTY('FilestreamShareName') ShareName,
       SERVERPROPERTY('FilestreamEffectiveLevel') EffectiveLevel,
       SERVERPROPERTY('FilestreamConfiguredLevel') ConfiguredLevel
```

This returns

ShareName	EffectiveLevel	ConfiguredLevel
AUGUSTUS	3	3

Next, I will create a new database that will have a filegroup containing FILESTREAM data.

Tip See Chapter 22 for more on the CREATE DATABASE command.

Unlike regular file/filegroup assignments in CREATE DATABASE, I will associate a filegroup with a specific path, and also designate the name of the folder that will be created by SQL Server on the file system and will contain all FILESTREAM files associated with the database:

```
USE master
GO

CREATE DATABASE PhotoRepository ON PRIMARY
( NAME = N'PhotoRepository',
  FILENAME = N'C:\Apress\MDF\PhotoRepository.mdf' ,
  SIZE = 3048KB ,
  FILEGROWTH = 1024KB ),
  FILEGROUP FS_PhotoRepository CONTAINS FILESTREAM
( NAME = 'FG_PhotoRepository',
  FILENAME = N'C:\Apress\FILESTREAM')
LOG ON
( NAME = N'PhotoRepository_log',
  FILENAME = N'C:\Apress\LDF\PhotoRepository_log.ldf' ,
  SIZE = 1024KB ,
  FILEGROWTH = 10%)
GO
```

Now I can create a new table that will be used to store photos for book covers. I will designate the BookPhotoFile column as a varbinary(max) data type, followed by the FILESTREAM attribute:

```
USE PhotoRepository
GO

CREATE TABLE dbo.BookPhoto
( BookPhotoID uniqueidentifier ROWGUIDCOL NOT NULL PRIMARY KEY,
  BookPhotoNM varchar(50) NOT NULL,
  BookPhotoFile varbinary(max) FILESTREAM)
GO
```

Now that the table is created, I can INSERT a new row using the regular INSERT command and importing a file using OPENROWSET (demonstrated in the previous recipe):

```
INSERT dbo.BookPhoto
(BookPhotoID, BookPhotoNM, BookPhotoFile)
SELECT NEWID(),
       'SQL Server 2008 Transact-SQL Recipes cover',
```

```
BulkColumn
FROM OPENROWSET(BULK
'C:\Apress\TSQL2008Recipes.bmp', SINGLE_BLOB) AS x
```

If I look under the C:\Apress\FILESTREAM directory, I will see a new subdirectory and a new file. In this case, on my server, I see a new file called 00000012-000000e1-0002 under the path C:\Apress\FILESTREAM\33486315-2ca1-43ea-a50e-0f84ad8c3fa6\e2f310f3-cd21-4f29-acd1-a0a3ffb1a681. Files created using FILESTREAM should only be accessed within the context of T-SQL and the associated Win32 APIs.

After inserting the value, I will now issue a SELECT to view the contents of the table:

```
SELECT BookPhotoID, BookPhotoNM, BookPhotoFile
FROM dbo.BookPhoto
```

This returns

BookPhotoID	BookPhotoNM	BookPhotoFile
236E5A69-53B3-4CB6-9F11- EF056082F542	SQL Server 2008 T-SQL Recipes cover	0x424D3656000000000000360400002800 0007D000000A4000000010008000000000 00520000000000000000000000100000001 0000276B8E0026B0ED005B5D6900EEEEEE00 528CA2000E0A0B001C597900B3B3B3008B8A 8D00D1D1D1002AC6FF002394C7002280AB00 2C2A2B00193F560066ADB0025A4DC001128 34005E

Now assuming I have an application that uses OLEDB to query the SQL Server instance, I need to now collect the appropriate information about the file system file in order to stream it using my application.

I'll begin by opening up a transaction and using the new PathName() method of the varbinary column to retrieve the logical path name of the file:

```
BEGIN TRAN

SELECT BookPhotoFile.PathName()
FROM dbo.BookPhoto
WHERE BookPhotoNM = 'SQL Server 2008 Transact-SQL Recipes cover'
```

This returns

```
\\CAESAR\AUGUSTUS\v1\PhotoRepository\dbo\BookPhoto\BookPhotoFile\
236E5A69-53B3-4CB6-9F11-EF056082F542
```

Next, I need to retrieve the transaction token, which is also needed by the Win 32 API:

```
SELECT GET_FILESTREAM_TRANSACTION_CONTEXT()
```

This returns

```
0x57773034AFA62746966EE30DAE70B344
```

After I have retrieved this information, the application can use the OpenSQLFileStream API with the path and transaction token to perform functions such as ReadFile and WriteFile and then close the handle to the file.

After the application is finished with its work, I can either roll back or commit the transaction:

```
COMMIT TRAN
```

If I wish to delete the file, I can set the column value to NULL:

```
UPDATE dbo.BookPhoto
SET BookPhotoFile = NULL
WHERE BookPhotoNM = 'SQL Server 2008 Transact-SQL Recipes cover'
```

You may not see the underlying file on the file system removed right away; however, it will be removed eventually by the garbage collector process.

How It Works

In this recipe, I demonstrated how to use the new SQL Server 2008 FILESYSTEM attribute of the varbinary(max) data type to store unstructured data on the file system. This enables SQL Server functionality to control transactions within SQL Server and recoverability (files get backed up with BACKUP and restored with RESTORE), while also being able to take advantage of high-speed streaming performance using Win 32 APIs.

In this recipe, I started off by checking whether FILESTREAM was enabled on the SQL Server instance. After that, I created a new database, designating the location of the FILESTREAM filegroup and file name (which is actually a path and not a file):

```
...
FILEGROUP FS_PhotoRepository CONTAINS FILESTREAM
(NAME = 'FG_PhotoRepository',
 FILENAME = N'C:\Apress\FILESTREAM')
```

Keep in mind that the path up to the last folder has to exist, but the last folder referenced cannot exist. For example, C:\Apress\ existed on my server; however, FILESTREAM can't exist prior to the database creation.

After creating the database, I then created a new table to store book cover images. For the BookPhotoFile column, I designated the varbinary(max) type followed by the FILESTREAM attribute:

```
...
BookPhotoFile varbinary(max) FILESTREAM
...
```

Had I left off the FILESTREAM attribute, any varbinary data stored would have been contained within the database data file, and not stored on the file system. The column maximum size would also have been capped at 2GB.

Next, I inserted a new row into the table that held the BMP file of the *SQL Server 2008 Transact-SQL Recipes* book cover. The varbinary(max) value was generated using the OPENROWSET technique I demonstrated in the previous recipe:

```
INSERT dbo.BookPhoto
(BookPhotoID, BookPhotoNM, BookPhotoFile)
SELECT NEWID(),
       'SQL Server 2008 Transact-SQL Recipes cover',
       BulkColumn
FROM OPENROWSET(BULK
 'C:\Apress\TSQL2008Recipes.bmp', SINGLE_BLOB) AS x
```

From an application perspective, I needed a couple of pieces of information in order to take advantage of streaming capabilities using Win 32 APIs. I started off by opening up a new transaction:

```
BEGIN TRAN
```

After that, I referenced the path name of the stored file using the `PathName()` method:

```
SELECT BookPhotoFile.PathName()
...
```

This function returned a path as a token, which the application can then use to grab a Win32 handle and perform operations against the value.

Next, I called the `GET_FILESTREAM_TRANSACTION_CONTEXT` function to return a token representing the current session's transaction context:

```
SELECT GET_FILESTREAM_TRANSACTION_CONTEXT()
```

This was a token used by the application to bind file system operations to an actual transaction.

After that, I committed the transaction and then demonstrated how to “delete” the file by updating the `BookPhotoFile` column to `NULL` for the specific row I had added earlier. Keep in mind that deleting the actual row would serve the same purpose (deleting the file on the file system).

Assigning and Modifying Database Values “in Place”

SQL Server 2008 introduces new compound assignment operators beyond the standard equality (`=`) operator that allow you to both assign and modify the outgoing data value. These operators are similar to what you would see in the C and Java languages. New assignment operators include the following:

- `+=` (add, assign)
- `-=` (subtract, assign)
- `*=` (multiply, assign)
- `/=` (divide, assign)
- `|=` (bitwise |, assign)
- `^=` (bitwise exclusive OR, assign)
- `&=` (bitwise &, assign)
- `%=` (modulo, assign)

This recipe will demonstrate modifying base pay amounts using assignment operators. I'll start by creating a new table and populating it with a few values:

```
USE AdventureWorks
GO

CREATE TABLE HumanResources.EmployeePayScale
    (EmployeePayScaleID int NOT NULL PRIMARY KEY IDENTITY(1,1),
    BasePayAMT numeric(9,2) NOT NULL,
    ModifiedDate datetime NOT NULL DEFAULT GETDATE())
GO

-- Using new multiple-row insert functionality
INSERT HumanResources.EmployeePayScale
(BasePayAMT)
VALUES
    (30000.00),
    (40000.00),
    (50000.00),
    (60000.00)
```

Next, I'll double-check the initial value of a specific pay scale row:

```
SELECT BasePayAMT
FROM HumanResources.EmployeePayScale
WHERE EmployeePayScaleID = 4
```

This returns

```
BasePayAMT
60000.00
```

Before SQL Server 2008, if I wanted to modify a value within an UPDATE based on the row's existing value, I would need to do something like the following:

```
UPDATE HumanResources.EmployeePayScale
SET BasePayAMT = BasePayAMT + 10000
WHERE EmployeePayScaleID = 4
```

Querying that row, I see that the base pay amount has increased by 10,000:

```
SELECT BasePayAMT
FROM HumanResources.EmployeePayScale
WHERE EmployeePayScaleID = 4
```

This returns

```
BasePayAMT
70000.00
```

Now I'll start experimenting with the assignment operators. This new feature allows me to simplify my code—assigning values in place without having to include another column reference in the value expression.

In this example, the base pay amount is increased by another 10,000 dollars:

```
UPDATE HumanResources.EmployeePayScale
SET BasePayAMT += 10000
WHERE EmployeePayScaleID = 4
```

```
SELECT BasePayAMT
FROM HumanResources.EmployeePayScale
WHERE EmployeePayScaleID = 4
```

This returns

```
BasePayAMT
80000.00
```

Next, the base pay amount is multiplied by 2:

```
UPDATE HumanResources.EmployeePayScale
SET BasePayAMT *= 2
WHERE EmployeePayScaleID = 4
```

```
SELECT BasePayAMT
FROM HumanResources.EmployeePayScale
WHERE EmployeePayScaleID = 4
```

This returns

```
BasePayAMT
160000.00
```

How It Works

Assignment operators help you modify values with a minimal amount of coding. In this recipe, I demonstrated using the add/assign operator:

```
SET BasePayAMT += 10000
```

and the multiply/assign operator:

```
SET BasePayAMT *= 2
```

The expressions designated the column name to be modified on the left, followed by the assignment operator, and then associated data value to be used with the operator. Keep in mind that this functionality isn't limited to UPDATE statements; you can use this new functionality when assigning values to variables.

DELETE

The simple syntax for DELETE is as follows:

```
DELETE [FROM] table_or_view_name
WHERE search_condition
```

The arguments of this command are described in Table 2-7.

Table 2-7. *The DELETE Command Arguments*

Argument	Description
table_or_view_name	This specifies the name of the table or updateable view that you are deleting rows from.
search_condition	The search condition(s) in the WHERE clause defines which rows will be deleted from the table or updateable view.

Deleting Rows

In this recipe, I show you how to use the DELETE statement to remove one or more rows from a table. First, take an example table that is populated with rows:

```
SELECT *
INTO Production.Example_ProductProductPhoto
FROM Production.ProductProductPhoto
```

This returns

```
(504 row(s) affected)
```

Next, *all* rows are deleted from the table:

```
DELETE Production.Example_ProductProductPhoto
```

This returns

```
(504 row(s) affected)
```

This next example demonstrates using DELETE with a WHERE clause. Let's say that the relationship of keys between two tables gets dropped, and the users were able to delete data from the primary key table and not the referencing foreign key tables (see Chapter 4 for a review of primary and foreign keys). Only rows missing a corresponding entry in the Product table are deleted from the example product photo table. In this example, no rows meet this criteria:

```
-- Repopulate the Example_ProductProductPhoto table
INSERT Production.Example_ProductProductPhoto
SELECT *
FROM Production.ProductProductPhoto
```

```
DELETE Production.Example_ProductProductPhoto
WHERE ProductID NOT IN
  (SELECT ProductID
   FROM Production.Product)
```

This third example demonstrates the same functionality of the previous example, only the DELETE has been rewritten to use a FROM clause instead of a subquery:

```
DELETE Production.ProductProductPhoto
FROM Production.Example_ProductProductPhoto ppp
LEFT OUTER JOIN Production.Product p ON
  ppp.ProductID = p.ProductID
WHERE p.ProductID IS NULL
```

How It Works

In the first example of the recipe, all rows were deleted from the Example_ProductProductPhoto table:

```
DELETE Production.Example_ProductProductPhoto
```

This is because there was no WHERE clause to specify which rows would be deleted.

In the second example, the WHERE clause was used to specify rows to be deleted based on a subquery lookup to another table:

```
WHERE ProductID NOT IN
  (SELECT ProductID
   FROM Production.Product)
```

The third example used a LEFT OUTER JOIN instead of a subquery, joining the ProductID of the two tables:

```
FROM Production.Example_ProductProductPhoto ppp
LEFT OUTER JOIN Production.Product p ON
  ppp.ProductID = p.ProductID
```

Because the same object that is being deleted from Production.ProductProductPhoto is also the same object in the FROM clause, and since there is only *one* reference to that table in the FROM clause,

it is assumed that rows identified in the FROM and WHERE clause will be one and the same—it will be associated to the rows deleted from the `Production.ProductProductPhoto` table.

Because a LEFT OUTER JOIN was used, if any rows did *not* match between the left and right tables, the fields selected from the right table would be represented by NULL values. Thus, to show rows in `Production.Example_ProductProductPhoto` that don't have a matching `ProductID` in the `Production.Product` table, you can qualify the `Production.Product` as follows:

```
WHERE p.ProductID IS NULL
```

Any rows without a match to the `Production.Product` table will be deleted from the `Production.Example_ProductProductPhoto` table.

Truncating a Table

In this recipe, I show you how to delete rows from a table in a minimally logged fashion (hence, much quicker if you have very large tables). Generally, you should use DELETE for operations that should be fully logged; however, for test or throwaway data, this is a fast technique for removing the data. “Minimal logging” references how much recoverability information is written to the database's transaction log (see Chapter 22). To achieve this, use the TRUNCATE command.

The syntax is as follows:

```
TRUNCATE TABLE table_name
```

This command takes just the table name to truncate. Since it always removes *all* rows from a table, there is no FROM or WHERE clause, as this recipe demonstrates:

```
-- First populating the example
SELECT *
INTO Sales.Example_Store
FROM Sales.Store

-- Next, truncating ALL rows from the example table
TRUNCATE TABLE Sales.Example_Store
```

Next, the table's row count is queried:

```
SELECT COUNT(*)
FROM Sales.Example_Store
```

This returns

```
0
```

How It Works

The TRUNCATE TABLE statement, like the DELETE statement, can delete rows from a table. TRUNCATE TABLE deletes rows faster than DELETE, because it is minimally logged. Unlike DELETE however, the TRUNCATE TABLE removes ALL rows in the table (no WHERE clause).

Although TRUNCATE TABLE is a faster way to delete rows, you can't use it if the table columns are referenced by a foreign key constraint (see Chapter 4 for more information on foreign keys), if the table is published using transactional or merge replication, or if the table participates in an indexed view (see Chapter 7 for more information). Also, if the table has an IDENTITY column, keep in mind that the column will be reset to the seed value defined for the column (if no seed was explicitly set, it is set to 1).

Advanced Data Modification Techniques

These next two recipes will demonstrate more advanced data modification techniques. Specifically, I'll demonstrate how to improve the throughput of data modifications by "chunking" them into smaller sets.

I'll also demonstrate the new SQL Server 2008 MERGE command, which you can use to efficiently apply changes to a target table based on the data in a table source without having to designate multiple DML statements.

Chunking Data Modifications with TOP

I demonstrated using TOP in Chapter 1. TOP can also be used in DELETE, INSERT, or UPDATE statements as well. This recipe further demonstrates using TOP to "chunk" data modifications, meaning instead of executing a very large operation in a single statement call, you can break the modification into smaller pieces, potentially increasing performance and improving database concurrency for larger, frequently accessed tables. This technique is often used for large data loads to reporting or data warehouse applications.

Large, single-set updates can cause the database transaction log to grow considerably. When processing in chunks, each chunk is committed after completion, allowing SQL Server to potentially reuse that transaction log space. In addition to transaction log space, on a very large data update, if the query must be cancelled, you may have to wait a long time while the transaction rolls back. With smaller chunks, you can continue with your update more quickly. Also, chunking allows more concurrency against the modified table, allowing user queries to jump in, instead of waiting several minutes for a large modification to complete.

In this recipe, I show you how to modify data in blocks of rows in multiple executions, instead of an entire result set in one large transaction. First, I create an example deletion table for this recipe:

```
USE AdventureWorks
GO

SELECT *
INTO Production.Example_BillofMaterials
FROM Production.BillofMaterials
```

Next, all rows will be deleted from the table in 500-row chunks:

```
WHILE (SELECT COUNT(*)FROM Production.Example_BillofMaterials)> 0
BEGIN

    DELETE TOP(500)
    FROM Production.Example_BillofMaterials

END
```

This returns

```
(500 row(s) affected)
(500 row(s) affected)
(500 row(s) affected)
(500 row(s) affected)
(500 row(s) affected)
(179 row(s) affected)
```

How It Works

In this example, I used a `WHILE` condition to keep executing the `DELETE` while the count of rows in the table was greater than zero (see Chapter 9 for more information on `WHILE`):

```
WHILE (SELECT COUNT(*)FROM Production.Example_BillofMaterials)> 0
BEGIN
```

Next was the `DELETE`, followed by the `TOP` clause, and the row limitation in parentheses:

```
DELETE TOP(500)
FROM Production.BillofMaterials
```

This recipe didn't use a `WHERE` clause, so no filtering was applied, and *all* rows were deleted from the table—but only in 500-row chunks. Once the `WHILE` condition no longer evaluated to `TRUE`, the loop ended. After executing, the row counts affected in each batch were displayed. The first five batches deleted 500 rows, and the last batch deleted the remaining 179 rows.

This “chunking” method can be used with `INSERTs` and `UPDATEs` too. For `INSERT` and `UPDATE`, the `TOP` clause follows right after the `INSERT` and `UPDATE` keyword, for example:

```
INSERT TOP(100)
...

UPDATE TOP(25)
...
```

The expanded functionality of `TOP` (beyond just `SELECT`) adds a new technique for managing large data modifications against a table. By reducing the size of large modifications, you can improve database concurrency by reducing the time that locks are held during the operation (leaving small windows for other sessions), and also help manage the size of the transaction log (more commits, instead of one single commit for a gigantic transaction).

Executing INSERTs, UPDATEs, and DELETEs in a Single Statement

SQL Server 2008 introduces the `MERGE` command to efficiently apply changes to a target table based on the data in a table source. If you've ever had to load and incrementally modify relational data warehouses or operational data stores based on incoming data from external data sources, you'll find this technique to be a big improvement over previous methods.

Rather than create multiple data modification statements, you can instead point `MERGE` to your target and source tables, defining what actions to take when search conditions find a match, when the target table does not have a match, or when the source table does not have a match. Based on these matching conditions, you can designate whether or not a `DELETE`, `INSERT`, or `UPDATE` operation takes place (again, within the same statement).

This recipe will demonstrate applying changes to a production table based on data that exists in a staging table (presumably staged data from an external data source). I'll start off by creating a production table that tells me whether or not a corporate housing unit is available for renting. If the `IsRentedIND` is 0, the unit is not available. If it is 1, it is available:

```
CREATE TABLE HumanResources.CorporateHousing
(CorporateHousingID int NOT NULL PRIMARY KEY IDENTITY(1,1),
 UnitNBR int NOT NULL,
 IsRentedIND bit NOT NULL,
 ModifiedDate datetime NOT NULL DEFAULT GETDATE())
GO
```

```
-- Insert existing units
INSERT HumanResources.CorporateHousing
(UnitNBR, IsRentedIND)
VALUES
(1, 0),
(24, 1),
(39, 0),
(54, 1)
```

In this scenario, I receive periodic data feeds that inform me of rental status changes for corporate units. Units can shift from rented to not rented. New units can be added based on contracts signed, and existing units can be removed due to contract modifications or renovation requirements. So for this recipe, I'll create a staging table that will receive the current snapshot of corporate housing units from the external data source. I'll also populate it with the most current information:

```
CREATE TABLE dbo.StagingCorporateHousing
    (UnitNBR int NOT NULL,
     IsRentedIND bit NOT NULL)
GO

INSERT dbo.StagingCorporateHousing
(UnitNBR, IsRentedIND)
VALUES
-- UnitNBR "1" no longer exists
(24, 0), -- UnitNBR 24 has a changed rental status
(39, 1), -- UnitNBR 39 is the same
(54, 0), -- UnitNBR 54 has a change status
(92, 1) -- UnitNBR 92 is a new unit, and isn't in production yet
```

Now my objective is to modify the target production table so that it reflects the most current data from our data source. If a new unit exists in the staging table, I want to add it. If a unit number exists in the production table but not the staging table, I want to delete the row. If a unit number exists in both the staging and production tables, but the rented status is different, I want to update the production (target) table to reflect the changes.

I'll start by looking at the values of production before the modification:

```
-- Before the MERGE
SELECT CorporateHousingID, UnitNBR, IsRentedIND
FROM HumanResources.CorporateHousing
```

This returns

CorporateHousingID	UnitNBR	IsRentedIND
1	1	0
2	24	1
3	39	0
4	54	1

Next, I'll modify the production table per my business requirements:

```
MERGE INTO HumanResources.CorporateHousing p
USING dbo.StagingCorporateHousing s
ON p.UnitNBR = s.UnitNBR
WHEN MATCHED AND s.IsRentedIND <> p.IsRentedIND THEN
UPDATE SET IsRentedIND = s.IsRentedIND
WHEN NOT MATCHED BY TARGET THEN
```

```
INSERT (UnitNBR, IsRentedIND) VALUES (s.UnitNBR, s.IsRentedIND)
WHEN NOT MATCHED BY SOURCE THEN
DELETE;
```

This returns

(5 row(s) affected)

Next, I'll check the "after" results of the production table:

```
-- After the MERGE
SELECT CorporateHousingID, UnitNBR, IsRentedIND
FROM HumanResources.CorporateHousing
```

This returns

CorporateHousingID	UnitNBR	IsRentedIND
2	24	0
3	39	1
4	54	0
5	92	1

How It Works

In this recipe, I demonstrated how to apply INSERT/UPDATE/DELETE modifications using a MERGE statement. The MERGE command allowed me to modify a target table based on the expression validated against a source staging table.

In the first line of the MERGE command, I designated the target table where I will be applying the data modifications:

```
MERGE INTO HumanResources.CorporateHousing p
```

On the second line, I identified the data source that will be used to compare the data against the target table. This source could have also been based on a derived or linked server table:

```
USING dbo.StagingCorporateHousing s
```

Next, I defined how I am joining these two data sources. In this case, I am using what is essentially a natural key of the data. This natural key is what uniquely identifies the row both in the source and target tables:

```
ON p.UnitNBR = s.UnitNBR
```

Next, I defined what happens when there is a match between the unit numbers by designating WHEN MATCHED. I also added an addition search condition, which indicates that if the rental indicator doesn't match, the rental indicator should be changed to match the staging data:

```
WHEN MATCHED AND s.IsRentedIND <> p.IsRentedIND THEN
UPDATE SET IsRentedIND = s.IsRentedIND
```

Next, I evaluated what happens when there is not a match from the source to the target table—for example, if the source table has a value of 92 for the unit number, but the target table does not have such a row. When this occurs, I directed this command to add the missing row to the target table:

```
WHEN NOT MATCHED BY TARGET THEN
INSERT (UnitNBR, IsRentedIND) VALUES (s.UnitNBR, s.IsRentedIND)
```


94 CHAPTER 2 ■ PERFORM, CAPTURE, AND TRACK DATA MODIFICATIONS

```
SELECT DeletedName,
       InsertedName
FROM @ProductChanges
```

This query returns

```
DeletedName      InsertedName
HL Spindle/Axle  HL Spindle/Axle XYZ
```

This next example uses OUTPUT for a DELETE operation. First, I'll create a demonstration table to hold the data:

```
SELECT *
INTO Sales.Example_SalesTaxRate
FROM Sales.SalesTaxRate
```

Next, I create a table variable to hold the data, delete rows from the table, and then select from the table variable to see which rows were deleted:

```
DECLARE @SalesTaxRate TABLE(
    [SalesTaxRateID] [int] NOT NULL,
    [StateProvinceID] [int] NOT NULL,
    [TaxType] [tinyint] NOT NULL,
    [TaxRate] [smallmoney] NOT NULL,
    [Name] [dbo].[Name] NOT NULL,
    [rowguid] [uniqueidentifier] ,
    [ModifiedDate] [datetime] NOT NULL )
```

```
DELETE Sales.Example_SalesTaxRate
OUTPUT DELETED.*
INTO @SalesTaxRate
```

```
SELECT SalesTaxRateID,
       Name
FROM @SalesTaxRate
```

This returns the following abridged results:

```
SalesTaxRateID      Name
1                   Canadian GST + Alberta Provincial Tax
2                   Canadian GST + Ontario Provincial Tax
3                   Canadian GST + Quebec Provincial Tax
4                   Canadian GST
...
27                  Washington State Sales Tax
28                  Taxable Supply
29                  Germany Output Tax
30                  France Output Tax
31                  United Kingdom Output Tax
```

(29 row(s) affected)

In the third example, I'll demonstrate using an INSERT with OUTPUT. A new row is inserted into a table, and the operation is captured to a table variable table:

```

DECLARE @NewDepartment TABLE
  (DepartmentID smallint NOT NULL,
   Name nvarchar(50) NOT NULL,
   GroupName nvarchar(50) NOT NULL,
   ModifiedDate datetime NOT NULL)

INSERT HumanResources.Department
(Name, GroupName)
OUTPUT INSERTED.*
INTO @NewDepartment
VALUES ('Accounts Receivable', 'Accounting')

SELECT DepartmentID,
       ModifiedDate
FROM @NewDepartment

```

This returns

DepartmentID	ModifiedDate
18	2007-09-15 08:38:28.833

How It Works

The first example used a temporary table variable to hold the OUTPUT results (see Chapter 4 for more information on temporary table variables):

```

DECLARE @ProductChanges TABLE
  (DeletedName nvarchar(50),
   InsertedName nvarchar(50))

```

Next, the first part of the UPDATE changed the product name to HL Spindle/Axle XYZ:

```

UPDATE Production.Product
SET Name = 'HL Spindle/Axle XYZ'

```

After the SET clause, but *before* the WHERE clause, the OUTPUT defined which columns to return:

```

OUTPUT DELETED.Name,
       INSERTED.Name

```

Like DML triggers (covered in Chapter 12), two “virtual” tables exist for the OUTPUT to use—INSERTED and DELETED—both of which hold the original and modified values for the updated table. The INSERTED and DELETED virtual tables share the same column names of the modified table—in this case returning the original name (DELETED.Name) and the new name (INSERTED.Name).

The values of this OUTPUT were placed into the temporary table variable by using INTO, followed by the table name:

```

INTO @ProductChanges

```

The UPDATE query qualified that only ProductID 524 would be modified to the new name:

```

WHERE ProductID = 524

```

After the update, a query was executed against the @ProductChanges temporary table variable to show the before/after changes:

```

SELECT DeletedName,
       InsertedName
FROM @ProductChanges

```

The DELETE and INSERT examples using OUTPUT were variations on the first example, where OUTPUT pushes the deleted rows (for DELETE) or the inserted rows (for INSERT) into a table variable.

Asynchronously Capturing Table Data Modifications

SQL Server 2008 provides a built-in method for asynchronously tracking all data modifications that occur against your user tables without your having to code your own custom triggers or queries. Change Data Capture has minimal performance overhead and can be used for incremental updates of other data sources, for example, migrating changes made in the OLTP database to your data warehouse database. The next set of recipes will demonstrate how to use this new functionality.

To begin with, I'll create a new database that will be used to demonstrate this functionality:

```
IF NOT EXISTS (SELECT name
              FROM sys.databases
              WHERE name = 'TSQLRecipe_CDC_Demo')
BEGIN
    CREATE DATABASE TSQLRecipe_CDC_Demo
END
GO
```

In this first recipe, I'll demonstrate adding CDC to a table in the TSQLRecipe_CDC_Demo database. The first step is to validate whether the database is enabled for Change Data Capture:

```
SELECT is_cdc_enabled
FROM sys.databases
WHERE name = 'TSQLRecipe_CDC_Demo'
```

This returns

```
is_cdc_enabled
0
```

Change Data Capture is configured and managed using various stored procedures. In order to enable the database, I'll execute the `sys.dp_cdc_enable_db` stored procedure in the context of the TSQLRecipe_CDC_Demo database:

```
USE TSQLRecipe_CDC_Demo
GO

EXEC sys.sp_cdc_enable_db
GO
```

This returns

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

Next, I'll revalidate that Change Data Capture is enabled:

```
SELECT is_cdc_enabled
FROM sys.databases
WHERE name = 'TSQLRecipe_CDC_Demo'
```

This returns

```
is_cdc_enabled
1
```

Now that Change Data Capture is enabled, I can proceed with capturing changes for tables in the database by using the `sys.sp_cdc_enable_table` system stored procedure. The parameters of this stored procedure are described in Table 2-8.

Table 2-8. *sp_cdc_enable_table* Parameters

Parameter	Description
@source_schema	This parameter defines the schema of the object.
@source_name	This parameter specifies the table name.
@role_name	This option allows you to select the name of the user-defined role that will have permissions to access the CDC data.
@capture_instance	You can designate up to <i>two</i> capture instances for a single table. This comes in handy if you plan on altering the schema of a table already captured by CDC. You can alter the schema without affecting the original CDC (unless it is a data type change), create a new capture instance, track changes in two tables, and then drop the original capture instance once you are sure the new schema capture fits your requirements. If you don't designate the name, the default value is <code>schema_source</code> .
@supports_net_changes	When enabled, this option allows you to show just the latest change to the data within the LSN range selected. This option requires a primary key be defined on the table. If no primary key is defined, you can also designate a unique key in the <code>@index_name</code> option.
@index_name	This parameter allows you to designate the unique key on the table to be used by CDC if a primary key doesn't exist.
@captured_column_list	If you aren't interested in tracking all column changes, this option allows you to narrow down the list.
@filegroup_name	This option allows you to designate where the CDC data will be stored. For very large data sets, isolation on a separate filegroup may yield better manageability and performance.
@partition_switch	This parameter takes a TRUE or FALSE value designating whether or not a <code>ALTER TABLE... SWITCH PARTITION</code> command will be allowed against the CDC table (default is FALSE).

In this recipe, I would like to track all changes against the following new table:

```
USE TSQRRecipe_CDC_Demo
GO

CREATE TABLE dbo.Equipment
(EquipmentID int NOT NULL PRIMARY KEY IDENTITY(1,1),
EquipmentDESC varchar(100) NOT NULL,
LocationID int NOT NULL)
GO
```

I would like to be able to capture all changes made to rows, as well as return just the net changes for a row. For other options, I'll be going with the default:

```
EXEC sys.sp_cdc_enable_table
@source_schema = 'dbo',
@source_name = 'Equipment',
@role_name = NULL,
@capture_instance = NULL,
@supports_net_changes = 1,
```

```
@index_name = NULL,
@captured_column_list = NULL,
@filegroup_name = default,
@partition_switch = FALSE
```

The results of this procedure call indicate that two SQL Server Agent jobs were created (SQL Server Agent has to be running):

```
Job 'cdc.TSQLRecipe_CDC_Demo_capture' started successfully.
Job 'cdc.TSQLRecipe_CDC_Demo_cleanup' started successfully.
```

Two jobs, a capture and a cleanup, are created for each database that has CDC enabled for tables.

Tip Had CDC already been enabled for a table in the same database, the jobs would not have been re-created.

I can confirm that this table is now tracked by executing the following query:

```
SELECT is_tracked_by_cdc
FROM sys.tables
WHERE name = 'Equipment' and
      schema_id = SCHEMA_ID('dbo')
```

This returns

```
is_tracked_by_cdc
1
```

I can also validate the settings of your newly configured capture instance using the `sys.sp_cdc_help_change_data_capture` stored procedure:

```
EXEC sys.sp_cdc_help_change_data_capture 'dbo', 'Equipment'
```

This returns the following result set (presented in name/value pairs for formatting purposes):

```
source_schema      dbo
source_table       Equipment
capture_instance   dbo_Equipment
object_id          357576312
source_object_id   293576084
start_lsn          NULL
end_lsn            NULL
supports_net_changes 1
has_drop_pending   NULL
role_name          NULL
index_name         PK_Equipmen__344745994707859D
filegroup_name     NULL
create_date        2008-03-16 09:27:52.990
index_column_list  [EquipmentID]
captured_column_list [EquipmentID], [EquipmentDESC], [LocationID]
```

How It Works

In this recipe, I started off by enabling CDC capabilities for the database using `sp_cdc_enable_db`. Behind the scenes, enabling CDC for the database creates a new schema called `cdc` and a few new tables in the database, detailed in Table 2-9. You shouldn't need to query these tables directly, as there are system stored procedures and functions that can return the same data in a cleaner format.

Table 2-9. *CDC System Tables*

Table	Description
<code>cdc.captured_columns</code>	Returns the columns tracked for a specific capture instance.
<code>cdc.change_tables</code>	Returns tables created when CDC is enabled for a table. Use <code>sys.sp_cdc_help_change_data_capture</code> to query this information rather than query this table directly.
<code>cdc.ddl_history</code>	Returns rows for each DDL change made to the table, once CDC is enabled. Use <code>sys.sp_cdc_get_ddl_history</code> instead of querying this table directly.
<code>cdc.index_columns</code>	Returns index columns associated with the CDC-enabled table. Query <code>sys.sp_cdc_help_change_data_capture</code> to retrieve this information rather than querying this table directly.
<code>cdc.lsn_time_mapping</code>	Helps you map the log sequence number to transaction begin and end times. Again, avoid querying the table directly, and instead use the functions <code>sys.fn_cdc_map_lsn_to_time</code> and <code>sys.fn_cdc_map_time_to_lsn</code> .

I'll review how some of the more commonly used functions and procedures are used in upcoming recipes.

After enabling the database for CDC, I then added CDC tracking to a user table in the database using the `sp_cdc_enable_table` procedure. I designated the schema, name, and the net changes flag. All other options were left to the default values.

Once `sp_cdc_enable_table` was executed, because this was the first source table to be enabled in the database, two new SQL Agent jobs were created. One job was called `cdc.TSQLRecipe_CDC_Demo_capture`. This job is responsible for capturing changes made using replication log reader technology and is scheduled to start automatically when SQL Server starts and run continuously. The second job, `cdc.TSQLRecipe_CDC_Demo_cleanup`, is scheduled by default to run daily at 2 a.m. and cleans up data older than three days (72 hours) by default.

Executing `sys.sp_cdc_help_change_data_capture` allowed me to validate various settings of the capture instance, including the support of net changes, tracking columns, creation date, and primary key used to determine uniqueness of the rows.

Enabling CDC for a table also causes a new table to be created in the CDC schema. In this case, a new table called `cdc.dbo_Equipment_CT` was created automatically. This table has the same columns as the base table, along with five additional columns added to track LSN, operation, and updated column information. You shouldn't query this directly, but instead use functions as I'll demonstrate in the next recipe.

Querying All Changes from CDC Tables

Now that CDC is enabled for the database and a change capture instance is created for a table, I'll go ahead and start making changes to the table in order to demonstrate the functionality:

100 CHAPTER 2 ■ PERFORM, CAPTURE, AND TRACK DATA MODIFICATIONS

```

USE TSQLRecipe_CDC_Demo
GO

INSERT dbo.Equipment
(EquipmentDESC, LocationID)
VALUES ('Projector A', 22)

INSERT dbo.Equipment
(EquipmentDESC, LocationID)
VALUES ('HR File Cabinet', 3)

UPDATE dbo.Equipment
SET EquipmentDESC = 'HR File Cabinet 1'
WHERE EquipmentID = 2

DELETE dbo.Equipment
WHERE EquipmentID = 1

```

After making the changes, I can now view a history of what was changed using the CDC functions. Data changes are tracked at the log sequence number (LSN) level. An LSN is a record in the transaction log that uniquely identifies activity.

I will now pull the minimum and maximum LSN values based on the time range I wish to pull changes for. To determine LSN, I'll use the `sys.fn_cdc_map_time_to_lsn` function, which takes two input parameters, the relational operator, and the tracking time (there are other ways to do this, which I demonstrate later on in the chapter). The relational operators are as follows:

- Smallest greater than
- Smallest greater than or equal
- Largest less than
- Largest less than or equal

These operators are used in conjunction with the Change Tracking time period you specify to help determine the associated LSN value. For this recipe, I want the minimum and maximum LSN values between two time periods:

```

SELECT sys.fn_cdc_map_time_to_lsn
('smallest greater than or equal' , '2008-03-16 09:34:11') as BeginLSN

SELECT sys.fn_cdc_map_time_to_lsn
('largest less than or equal' , '2008-03-16 23:59:59') as EndLSN

```

This returns the following results (your actual LSN if you are following along will be different):

```

BeginLSN
0x0000001C000001020001

(1 row(s) affected)

EndLSN
0x0000001C000001570001

(1 row(s) affected)

```

I now have my LSN boundaries to detect changes that occurred during the desired time range.

My next decision is whether or not I wish to see all changes or just net changes. I can call the same functions demonstrated in the previous query in order to generate the LSN boundaries and populate them into variables for use in the `cdc.fn_cdc_get_all_changes_dbo_Equipment` function. As the name of that function suggests, I'll demonstrate showing *all* changes first:

```
DECLARE @FromLSN varbinary(10) =
    sys.fn_cdc_map_time_to_lsn
        ( 'smallest greater than or equal' , '2008-03-16 09:34:11')

DECLARE @ToLSN varbinary(10) =
    sys.fn_cdc_map_time_to_lsn
        ( 'largest less than or equal' , '2008-03-16 23:59:59')

SELECT
    __operation,
    __update_mask,
    EquipmentID,
    EquipmentDESC,
    LocationID
FROM cdc.fn_cdc_get_all_changes_dbo_Equipment
    (@FromLSN, @ToLSN, 'all')
```

This returns the following result set:

__operation	__update_mask	EquipmentID	EquipmentDESC	LocationID
2	0x07	1	Projector A	22
2	0x07	2	HR File Cabinet	3
4	0x02	2	HR File Cabinet 1	3
1	0x07	1	Projector A	22

This result set revealed all modifications made to the table. Notice that the function name, `cdc.fn_cdc_get_all_changes_dbo_Equipment`, was based on my CDC instance capture name for the source table. Also notice the values of `__operation` and `__update_mask`. The `__operation` values are interpreted as follows:

- 1 is a delete.
- 2 is an insert.
- 3 is the “prior” version of an updated row (use `all update old` option to see—I didn't use this in the prior query).
- 4 is the “after” version of an updated row.

The update mask uses bits to correspond to the capture column modified for an operation. I'll demonstrate how to translate these values in a separate recipe.

Moving forward in this current recipe, I could have also used the `all update old` option to show previous values of an updated row prior to the modification. I can also add logic to translate the values seen in the result set for the operation type. For example:

```
DECLARE @FromLSN varbinary(10) =
    sys.fn_cdc_map_time_to_lsn
        ( 'smallest greater than or equal' , '2008-03-16 09:34:11')

DECLARE @ToLSN varbinary(10) =
    sys.fn_cdc_map_time_to_lsn
        ( 'largest less than or equal' , '2008-03-16 23:59:59')
```

```

SELECT
  CASE __$operation
    WHEN 1 THEN 'DELETE'
    WHEN 2 THEN 'INSERT'
    WHEN 3 THEN 'Before UPDATE'
    WHEN 4 THEN 'After UPDATE'
  END Operation,
  __$update_mask,
  EquipmentID,
  EquipmentDESC,
  LocationID
FROM cdc.fn_cdc_get_all_changes_dbo_Equipment
(@FromLSN, @ToLSN, 'all update old')

```

This returns

Operation	__\$update_mask	EquipmentID	EquipmentDESC	LocationID
INSERT	0x07	1	Projector A	22
INSERT	0x07	2	HR File Cabinet	3
Before UPDATE	0x02	2	HR File Cabinet	3
After UPDATE	0x02	2	HR File Cabinet 1	3
DELETE	0x07	1	Projector A	22

How It Works

In this recipe, modifications were made against the CDC tracked table. Because the underlying CDC data is actually tracked by LSN, I needed to translate my min/max time range to the minimum and maximum LSNs that would include the data changes I was looking for. This was achieved using `sys.fn_cdc_map_time_to_lsn`.

Tip There is also a `sys.fn_cdc_map_lsn_to_time` function available to convert your tracked LSNs to temporal values.

Next, I executed the `cdc.fn_cdc_get_all_changes_dbo_Equipment` function, which allowed me to return all changes made for the LSN range I passed:

```

SELECT
  __$operation,
  __$update_mask,
  EquipmentID,
  EquipmentDESC,
  LocationID
FROM cdc.fn_cdc_get_all_changes_dbo_Equipment
(@FromLSN, @ToLSN, 'all')

```

For an ongoing incremental load, it may also make sense to store the beginning and ending LSN values for each load, and then use the `sys.fn_cdc_increment_lsn` function to increment the old upper bound LSN value to be your future lower bound LSN value for the next load (I'll demonstrate this in a later recipe).

In the last example of this recipe, I used the `all update old` parameter to return both before and after versions of rows from UPDATE statements, and also encapsulated the operation column in a CASE statement for better readability.

Querying Net Changes from CDC Tables

In the original CDC setup recipe, `sp_cdc_enable_table_change_data_capture` was executed with `@supports_net_changes = 1` for the source table. This means that I also have the option of executing the *net* changes version of the CDC procedure. The `fn_cdc_get_net_changes_` version of the stored procedure also takes a beginning and ending LSN value; however, the third parameter differs in the row filter options:

- `all`, which returns the last version of a row without showing values in the update mask.
- `all with mask`, which returns the last version of the row along with the update mask value (the next recipe details how to interpret this mask).
- `all with merge`, which returns the final version of the row as either a delete or a merge operation (either an insert or update). Inserts and updates are not broken out.

The following recipe demonstrates showing net changes without displaying the update mask. I'll start by issuing a few new data modifications:

```
INSERT dbo.Equipment
(EquipmentDESC, LocationID)
VALUES
('Portable White Board', 18)
```

```
UPDATE dbo.Equipment
SET LocationID = 1
WHERE EquipmentID = 3
```

Next, I track the net effect of my changes using the following query:

```
DECLARE @FromLSN varbinary(10) =
sys.fn_cdc_map_time_to_lsn
('smallest greater than or equal', '2008-03-16 09:45:00')

DECLARE @ToLSN varbinary(10) =
sys.fn_cdc_map_time_to_lsn
('largest less than or equal', '2008-03-16 23:59:59')

SELECT
CASE __operation
WHEN 1 THEN 'DELETE'
WHEN 2 THEN 'INSERT'
WHEN 3 THEN 'Before UPDATE'
WHEN 4 THEN 'After UPDATE'
WHEN 5 THEN 'MERGE'
END Operation,
__update_mask,
EquipmentID,
EquipmentDESC,
LocationID
FROM cdc.fn_cdc_get_net_changes_dbo_Equipment
(@FromLSN, @ToLSN, 'all with mask')
```

This returns

Operation	__update_mask	EquipmentID	EquipmentDESC	LocationID
INSERT	NULL	3	Portable White Board	1

How It Works

In this recipe, I used `cdc.fn_cdc_get_net_changes_dbo_Equipment` to return the net changes of rows between the specific LSN range. I first inserted a new row and then updated it. I queried `cdc.fn_cdc_get_net_changes_dbo_Equipment` to show the net change based on the LSN range. Although two changes were made, only one row was returned to reflect the final change needed, an INSERT operation that would produce the final state of the row.

Translating the CDC Update Mask

The update mask returned by the `cdc.fn_cdc_get_all_changes_` and `cdc.fn_cdc_get_net_changes_` functions allows you to determine which columns were affected by a particular operation. In order to interpret this value, however, you need the help of a couple of other CDC functions:

- `sys.fn_cdc_is_bit_set` is used to check whether a specific bit is set within the mask. Its first parameter is the ordinal position of the bit to check, and the second parameter is the update mask itself.
- `sys.fn_cdc_get_column_ordinal` is the function you can use in conjunction with `sys.fn_cdc_is_bit_set` to determine the ordinal position of the column for the table. This function's first parameter is the name of the capture instance. The second parameter is the name of the column.

In this recipe, I'll use both of these functions to help identify which columns were updated within the specific LSN boundary. First, I'll make two updates against two different rows:

```
UPDATE dbo.Equipment
SET EquipmentDESC = 'HR File Cabinet A1'
WHERE EquipmentID = 2
```

```
UPDATE dbo.Equipment
SET LocationID = 35
WHERE EquipmentID = 3
```

Now I'll issue a query to determine which columns have been changed using the update mask:

```
DECLARE @FromLSN varbinary(10) =
    sys.fn_cdc_map_time_to_lsn
        ( 'smallest greater than or equal' , '2008-03-16 10:02:00' )

DECLARE @ToLSN varbinary(10) =
    sys.fn_cdc_map_time_to_lsn
        ( 'largest less than or equal' , '2008-03-16 23:59:59' )

SELECT
    sys.fn_cdc_is_bit_set (
        sys.fn_cdc_get_column_ordinal (
            'dbo_Equipment' , 'EquipmentDESC' ),
        __update_mask) EquipmentDESC_Updated,
    sys.fn_cdc_is_bit_set (
        sys.fn_cdc_get_column_ordinal (
            'dbo_Equipment' , 'LocationID' ),
        __update_mask) LocationID_Updated,
    EquipmentID,
    EquipmentDESC,
    LocationID
```

```
FROM cdc.fn_cdc_get_all_changes_dbo_Equipment
    (@FromLSN, @ToLSN, 'all')
WHERE __operation = 4
```

This returns

EquipmentDESC_Updated	LocationID_Updated	EquipmentID	EquipmentDESC	LocationID
1	0	2	HR File Cabinet A1	3
0	1	3	Portable White Board	35

How It Works

In this recipe, I updated two rows. One update involved changing only the equipment description, and the second update involved changing the location ID.

In order to identify whether or not a bit is set, I used the following function call:

```
SELECT sys.fn_cdc_is_bit_set (
```

The first parameter of this function call is the ordinal position of the column I wish to check. In order to return this information, I used the following function call:

```
sys.fn_cdc_get_column_ordinal ( 'dbo_Equipment' , 'EquipmentDESC' )
```

The second parameter of `sys.fn_cdc_is_bit_set` is the update mask column name to be probed. I referenced this, along with an aliased name of the column in the query:

```
, __update_mask) EquipmentDESC_Updated,
```

I repeated this code for the `LocationID` in the next line of the query:

```
sys.fn_cdc_is_bit_set (sys.fn_cdc_get_column_ordinal
( 'dbo_Equipment' , 'LocationID' ), __update_mask) LocationID_Updated,
```

The rest of the query was standard, returning the change column values and querying the “all changes” CDC function:

```
DepartmentID,
Name,
GroupName
FROM cdc.fn_cdc_get_all_changes_dbo_Department
    (@FromLSN, @ToLSN, 'all')
```

Lastly, I qualified the query to only return type 4 rows, which are after versions of rows for an update operation:

```
WHERE __operation = 4
```

Working with LSN Boundaries

I've demonstrated how to determine the minimum and maximum LSN boundaries using `sys.fn_cdc_map_time_to_lsn`. However, you aren't limited to just using this function to define your boundaries. The following functions in this recipe can also be used to generate LSN values:

- `sys.fn_cdc_increment_lsn` allows you to return the next LSN number based on the input LSN number. So, for example, you could use this function to convert your last loaded upper bound LSN into your next lower bound LSN.

- `sys.fn_cdc_decrement_lsn` returns the prior LSN based on the input LSN number.
- `sys.fn_cdc_get_max_lsn` returns the largest LSN from the CDC data collected for your capture instance.
- `sys.fn_cdc_get_min_lsn` returns the oldest LSN from the CDC data collected for your capture instance.

The following recipe demonstrates retrieving LSN values from the CDC data collected for the `dbo.Equipment` table:

```
SELECT sys.fn_cdc_get_min_lsn ('dbo_Equipment') Min_LSN
```

```
SELECT sys.fn_cdc_get_max_lsn () Max_LSN
```

```
SELECT sys.fn_cdc_increment_lsn (sys.fn_cdc_get_max_lsn()) New_Lower_Bound_LSN
```

```
SELECT sys.fn_cdc_decrement_lsn (sys.fn_cdc_get_max_lsn())  
New_Lower_Bound_Minus_one_LSN
```

This returns the following (note that your results will be different):

```
Min_LSN
```

```
0x0000001C000001040014
```

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

```
Max_LSN
```

```
0x0000001E0000008B0001
```

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

```
New_Lower_Bound_LSN
```

```
0x0000001E0000008B0002
```

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

```
New_Lower_Bound_Minus_one_LSN
```

```
0x0000001E0000008B0000
```

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

How It Works

The new CDC functionality provides built-in methods for tracking changes to target tables in your database; however, you must still consider what logic you will use to capture time ranges for your Change Tracking. This recipe demonstrated methods you can use to retrieve the minimum and maximum available LSNs from the CDC database.

The `sys.fn_cdc_get_min_lsn` function takes the capture instance name as its input parameter, whereas `sys.fn_cdc_get_max_lsn` returns the maximum LSN at the database scope. The `sys.fn_cdc_increment_lsn` and `sys.fn_cdc_decrement_lsn` functions are used to increase and decrease the LSN based on the LSN you pass it. These functions allow you to create new boundaries for queries against the CDC data.

Disabling Change Data Capture from Tables and the Database

This recipe demonstrates how to remove Change Data Capture from a table. To do so, I'll execute the `sys.sp_cdc_disable_table` stored procedure. In this example, I will disable all Change Tracking from the table that may exist:

```
EXEC sys.sp_cdc_disable_table  
    'dbo', 'Equipment', 'all'
```

I can then validate that the table is truly disabled by executing the following query:

```
SELECT is_tracked_by_cdc  
FROM sys.tables  
WHERE name = 'Equipment' and  
    schema_id = SCHEMA_ID('dbo')
```

This returns

```
is_tracked_by_cdc  
0  
  
(1 row(s) affected)
```

To disable CDC for the database itself, I execute the following stored procedure:

```
EXEC sys.sp_cdc_disable_db
```

This returns

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

How It Works

The stored procedure `sys.sp_cdc_disable_table` is used to remove CDC from a table. The first parameter of this stored procedure designates the schema name, and the second parameter designates the table name. The last parameter designates whether you wish to remove all Change Tracking by designating `all` or instead specify the name of the capture instance.

To entirely remove CDC abilities from the database itself, I executed the `sys.sp_cdc_disable_db` procedure, which also removes the CDC schema and associated SQL Agent jobs.

Tracking Net Data Changes with Minimal Disk Overhead

CDC was intended to be used for asynchronous tracking of incremental data changes for data stores and warehouses and also provides the ability to detect intermediate changes to data. Unlike CDC, *Change Tracking* is a synchronous process that is part of the transaction of a DML operation itself (INSERT/UPDATE/DELETE) and is intended to be used for detecting net row changes with minimal disk storage overhead.

The synchronous behavior of Change Tracking allows for a transactionally consistent view of modified data, as well as the ability to detect data conflicts. Applications can use this functionality with minimal performance overhead and without the need to add supporting database object modifications (no custom change-detection triggers or table timestamps needed).

In this recipe, I'll walk through how to use the new Change Tracking functionality to detect DML operations. To begin with, I'll create a new database that will be used to demonstrate this functionality:

```

IF NOT EXISTS (SELECT name
               FROM sys.databases
               WHERE name = 'TSQLRecipeChangeTrackDemo')
BEGIN
    CREATE DATABASE TSQLRecipeChangeTrackDemo
END
GO

```

To enable Change Tracking functionality for the database, I have to configure the `CHANGE_TRACKING` database option. I also can configure how long changes are retained in the database and whether or not automatic cleanup is enabled. Configuring your retention period will impact how much Change Tracking is maintained for the database. Setting this value too high can impact storage. Setting it too low could cause synchronization issues with the other application databases if the remote applications do not synchronize often enough:

```

ALTER DATABASE TSQLRecipeChangeTrackDemo
SET CHANGE_TRACKING = ON
(CHANGE_RETENTION = 36 HOURS,
 AUTO_CLEANUP = ON)

```

A best practice when using Change Tracking is to enable the database for Snapshot Isolation. For databases and tables with significant DML activity, it will be important that you capture Change Tracking information in a consistent fashion—grabbing the latest version and using that version number to pull the appropriate data.

Caution Enabling Snapshot Isolation will result in additional space usage in `tempdb` due to row versioning generation. This can also increase overall I/O overhead.

Not using Snapshot Isolation can result in transactionally inconsistent change information:

```

ALTER DATABASE TSQLRecipeChangeTrackDemo
SET ALLOW_SNAPSHOT_ISOLATION ON
GO

```

I can confirm that I have properly enabled the database for Change Tracking by querying `sys.change_tracking_databases`:

```

SELECT DB_NAME(database_id) DBNM, is_auto_cleanup_on,
       retention_period, retention_period_units_desc
FROM sys.change_tracking_databases

```

This returns

DBNM	is_auto_cleanup_on	retention_period	retention_period_units_desc
TSQLRecipeChangeTrackDemo	1	36	HOURS

Now I will create a new table that will be used to demonstrate Change Tracking:

```

USE TSQLRecipeChangeTrackDemo
GO

CREATE TABLE dbo.BookStore
(BookStoreID int NOT NULL IDENTITY(1,1) PRIMARY KEY CLUSTERED,
 BookStoreNM varchar(30) NOT NULL,
 TechBookSection bit NOT NULL)
GO

```

Next, for each table that I wish to track changes for, I need to use the ALTER TABLE command with the CHANGE_TRACKING option. If I also want to track which columns were updated, I need to enable the TRACK_COLUMNS_UPDATED option, as demonstrated next:

```
ALTER TABLE dbo.BookStore
ENABLE CHANGE_TRACKING
WITH (TRACK_COLUMNS_UPDATED = ON)
```

I can validate which tables are enabled for Change Tracking by querying the sys.change_tracking_tables catalog view:

```
SELECT OBJECT_NAME(object_id) ObjNM,is_track_columns_updated_on
FROM sys.change_tracking_tables
```

This returns

ObjNM	is_track_columns_updated_on
BookStore	1

Now I will demonstrate Change Tracking by doing an initial population of the table with three new rows:

```
INSERT dbo.BookStore
(BookStoreNM, TechBookSection)
VALUES
('McGarnicles and Bailys', 1),
('Smith Book Store', 0),
('University Book Store',1)
```

One new function I can use for ongoing synchronization is the CHANGE_TRACKING_CURRENT_VERSION function, which returns the version number from the last committed transaction for the table. Each DML operation that occurs against a change-tracked table will cause the version number to increment. I'll be using this version number later on to determine changes:

```
SELECT CHANGE_TRACKING_CURRENT_VERSION ()
```

This returns

1

Also, I can use the CHANGE_TRACKING_MIN_VALID_VERSION function to check the minimum version available for the change-tracked table. If a disconnected application is not synchronized for a period of time exceeding the Change Tracking retention period, a full refresh of the application data would be necessary:

```
SELECT CHANGE_TRACKING_MIN_VALID_VERSION
( OBJECT_ID('dbo.BookStore') )
```

This returns

0

To detect changes, I can use the CHANGETABLE function. This function has two varieties of usage, using the CHANGES keyword to detect changes as of a specific synchronization version and using the VERSION keyword to return the latest Change Tracking version for a row.

I'll start off by demonstrating how CHANGES works. The following query demonstrates returning the latest changes to the BookStore table as of version 0. The first parameter is the name of the Change Tracking table, and the second parameter is the version number:

```
SELECT BookStoreID, SYS_CHANGE_OPERATION,
       SYS_CHANGE_VERSION
FROM CHANGETABLE
     (CHANGES dbo.BookStore, 0) AS CT
```

This returns the primary key of the table, followed by the DML operation type (I for INSERT, U for UPDATE, and D for DELETE), and the associated row version number (since all three rows were added for a single INSERT, they all share the same version number):

BookStoreID	SYS_CHANGE_OPERATION	SYS_CHANGE_VERSION
1	I	1
2	I	1
3	I	1

Caution When gathering synchronization information, use SET TRANSACTION ISOLATION LEVEL SNAPSHOT and BEGIN TRAN...COMMIT TRAN to encapsulate gathering of change information and associated current Change Tracking versions and minimum valid versions. Using Snapshot Isolation will allow for a transactionally consistent view of the Change Tracking data.

Now I'll modify the data a few more times in order to demonstrate Change Tracking further:

```
UPDATE dbo.BookStore
SET BookStoreNM = 'King Book Store'
WHERE BookStoreID = 1
```

```
UPDATE dbo.BookStore
SET TechBookSection = 1
WHERE BookStoreID = 2
```

```
DELETE dbo.BookStore
WHERE BookStoreID = 3
```

I'll check the latest version number:

```
SELECT CHANGE_TRACKING_CURRENT_VERSION ()
```

This is now incremented by three (there were three operations that acted against the data):

```
4
```

Now let's assume that an external application gathered information as of version 1 of the data. The following query demonstrates how to detect any changes that have occurred since version 1:

```
SELECT BookStoreID,
       SYS_CHANGE_VERSION,
       SYS_CHANGE_OPERATION,
       SYS_CHANGE_COLUMNS
FROM CHANGETABLE
     (CHANGES dbo.BookStore, 1) AS CT
```

This returns information on the rows that were modified since version 1, displaying the primary keys for the two updates I performed earlier and the primary key for the row I deleted:

BookStoreID	SYS_CHANGE_VERSION	SYS_CHANGE_OPERATION	SYS_CHANGE_COLUMNS
1	2	U	0x0000000002000000
2	3	U	0x0000000003000000
3	4	D	NULL

The `SYS_CHANGE_COLUMNS` column is a varbinary value that contains the columns that changed since the last version. To interpret this, I can use the `CHANGE_TRACKING_IS_COLUMN_IN_MASK` function, as I'll demonstrate next. This function takes two arguments, the column ID of the table column and the varbinary value to be evaluated. The following query uses this function to check whether the columns `BookStoreNM` and `TechBookSection` were modified:

```
SELECT BookStoreID,
       CHANGE_TRACKING_IS_COLUMN_IN_MASK(
         COLUMNPROPERTY(
           OBJECT_ID('dbo.BookStore'),'BookStoreNM', 'ColumnId') ,
           SYS_CHANGE_COLUMNS) IsChanged_BookStoreNM,
       CHANGE_TRACKING_IS_COLUMN_IN_MASK(
         COLUMNPROPERTY(
           OBJECT_ID('dbo.BookStore'),'TechBookSection', 'ColumnId') ,
           SYS_CHANGE_COLUMNS) IsChanged_TechBookSection
FROM CHANGETABLE
     (CHANGES dbo.BookStore, 1) AS CT
WHERE SYS_CHANGE_OPERATION = 'U'
```

This returns bit values of 1 for true and 0 for false regarding what columns were modified:

BookStoreID	IsChanged_BookStoreNM	IsChanged_TechBookSection
1	1	0
2	0	1

Next, I'll demonstrate that the `VERSION` argument of `CHANGETABLE` can be used to return the latest change version for each row. This version value can be stored and tracked by the application in order to facilitate Change Tracking synchronization:

```
SELECT bs.BookStoreID, bs.BookStoreNM, bs.TechBookSection,
       ct.SYS_CHANGE_VERSION
FROM dbo.BookStore bs
CROSS APPLY CHANGETABLE
     (VERSION dbo.BookStore, (BookStoreID), (bs.BookStoreID)) as ct
```

This returns the `SYS_CHANGE_VERSION` column along with the current column values for each row:

BookStoreID	BookStoreNM	TechBookSection	SYS_CHANGE_VERSION
1	King Book Store	1	2
2	Smith Book Store	1	3

Now I'll perform another `UPDATE` to demonstrate the version differences:

```
UPDATE dbo.BookStore
SET BookStoreNM = 'Kingsly Book Store',
    TechBookSection = 0
WHERE BookStoreID = 1
```

Next, I'll execute another query using CHANGETABLE:

```
SELECT bs.BookStoreID, bs.BookStoreNM, bs.TechBookSection,
       ct.SYS_CHANGE_VERSION
FROM dbo.BookStore bs
CROSS APPLY CHANGETABLE
         (VERSION BookStore, (BookStoreID), (bs.BookStoreID)) as ct
```

This shows that the row version of the row I just modified is now incremented to 5—but the other row that I did not modify remains at a version number of 2:

BookStoreID	BookStoreNM	TechBookSection	SYS_CHANGE_VERSION
1	Kingsly Book Store	0	5
2	Smith Book Store	1	3

I'll now check the current version number:

```
SELECT CHANGE_TRACKING_CURRENT_VERSION ()
```

This returns

```
5
```

The version number matches the latest change made to the table for the last committed transaction.

For the final part of this recipe, I will also demonstrate how to provide Change Tracking application context information with your DML operations. This will allow you to track which application made data modifications to which rows—which is useful information if you are synchronizing data across several data sources. In order to apply this data lineage, I can use the CHANGE_TRACKING_CONTEXT function. This function takes a single input parameter of context, which is a varbinary data type value representing the calling application.

I start off by declaring a variable to hold the application context information. I then use the variable within the CHANGE_TRACKING_CONTEXT function prior to an INSERT of a new row to the change-tracked table:

```
DECLARE @context varbinary(128) = CAST('Apress_XYZ' as varbinary(128));

WITH CHANGE_TRACKING_CONTEXT (@context)
INSERT dbo.BookStore
(BookStoreNM, TechBookSection)
VALUES
('Capers Book Store', 1)
```

Next, I will check for any changes that were made since version 5 (what I retrieved earlier on using CHANGE_TRACKING_CURRENT_VERSION):

```
SELECT BookStoreID,
       SYS_CHANGE_OPERATION,
       SYS_CHANGE_VERSION,
       CAST(SYS_CHANGE_CONTEXT as varchar) ApplicationContext
FROM CHANGETABLE
     (CHANGES dbo.BookStore, 5) AS CT
```

This returns the new row value that was inserted, along with the application context information that I converted from the SYS_CHANGE_CONTEXT column:

BookStoreID	SYS_CHANGE_OPERATION	SYS_CHANGE_VERSION	ApplicationContext
4	I	6	Apress_XYZ

How It Works

In this recipe, I demonstrated how to use Change Tracking in order to detect net row changes with minimal disk storage overhead. I started off by creating a new database and then using `ALTER DATABASE...SET CHANGE_TRACKING` to enable Change Tracking in the database. I also designated a 36-hour Change Tracking retention using the `CHANGE_RETENTION` and `AUTO_CLEANUP` options. I used the `sys.change_tracking_databases` catalog view to check the status of the change-tracked database.

I also enabled Snapshot Isolation for the database. This is a best practice, as you'll want to use Snapshot Isolation-level transactions when retrieving row change versions and the associated data from the change-tracked table.

Next, I created a new table and then used `ALTER TABLE...ENABLE CHANGE_TRACKING`. I designated that column-level changes also be tracked by enabling `TRACK_COLUMNS_UPDATED`. I validated the change-tracked status of the table by querying the `sys.change_tracking_tables` catalog view.

After that, I demonstrated several different functions that are used to retrieve Change Tracking data, including

- `CHANGE_TRACKING_CURRENT_VERSION`, which returns the version number from the last committed transaction for the table
- `CHANGE_TRACKING_MIN_VALID_VERSION`, which returns the minimum version available for the change-tracked table
- `CHANGETABLE` with `CHANGES`, to detect changes as of a specific synchronization version
- `CHANGE_TRACKING_IS_COLUMN_IN_MASK`, to detect which columns were updated from a change-tracked table
- `CHANGETABLE` with `VERSION`, to return the latest change version for a row
- `CHANGE_TRACKING_CONTEXT`, to store change context with a DML operation so you can track which application modified what data

Change Tracking as a feature set allows you to avoid having to custom-code your own net Change Tracking solution. This feature has minimal overhead and doesn't require schema modification in order to implement (no triggers or timestamps).

