http://www.sqlservercentral.com/articles/CDC/72020/
Printed 2011/01/17 11:43AM

Incremental Data Loading Using CDC

By Mark Murphy, 2010/12/27

Introduction

In my previous two articles, I described a technique for first querying, and then synchronizing two tables while tracking change history. In this third and final article in the series, I'll take the approach a step further by describing Change Data Capture (CDC), a feature introduced in SQL Server 2008 Enterprise Edition. CDC enables developers to set up incremental loads of data-warehouse fact tables that can execute in mere seconds -- instead of full loads that can take hours.

To illustrate CDC, I'll revisit and improve upon the sales data warehouse example I began in article two. The CDC-based data load design pattern retains the notion of a "slowly changing fact" table -- one that tracks all data changes in a point-in-time, effective-dated fashion. As you will see, the CDC-based approach generates the same differential records produced by the full-join approach, yet in a fraction of the time.

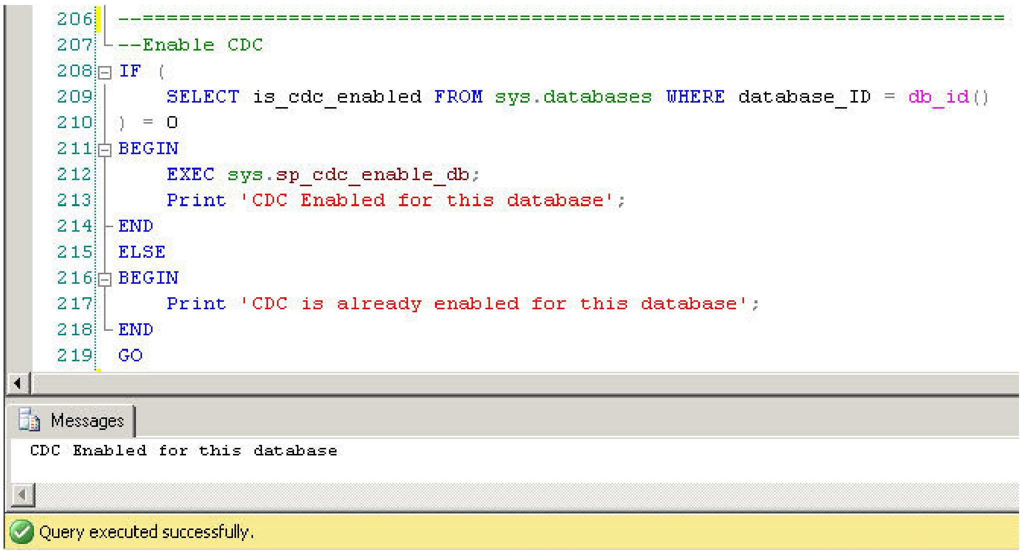
CDC Basics

When you apply CDC to a table, all DML (inserts, updates, and deletes) issued to the table are tracked in a second table automatically created by SQL Server. Internally, SQL Server tracks all DML to the base table by reading committed transactions from the transaction log, similar to the way it implements replication. This log reading happens in a background process and does not add additional work to the originating transaction. Naturally, there is an overhead performance cost to applying CDC, but this cost is only around 10%.

One downside of CDC is that it doesn't track which user made the DML call, so a workaround must be added to accomplish that -- such as capturing the current user's suser_sname() and storing it in a column via a trigger.

For an in-depth overview of CDC, please see this excellent introduction to CDC written by Pinal Dave.

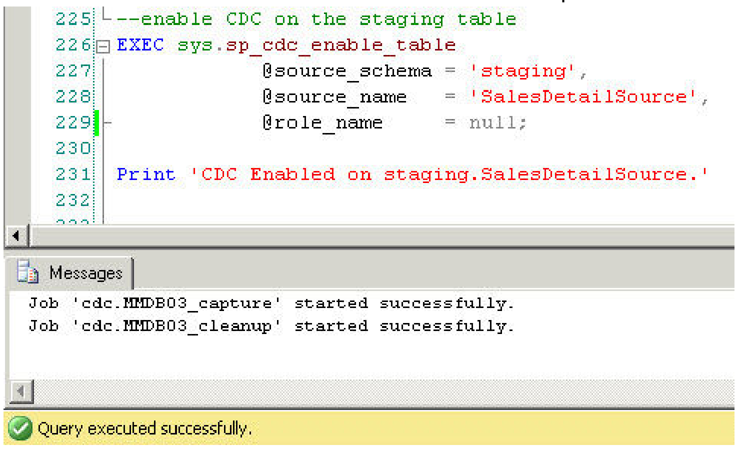
Let's jump into our example. Note that all the example code is attached at the bottom in the script file "cdc code.sql." To get started with CDC, you must first enable it on the database:



```
--==========================================================================
--Enable CDC
IF (
    SELECT is_cdc_enabled FROM sys.databases WHERE database_ID = db_id()
) = 0
BEGIN
    EXEC sys.sp_cdc_enable_db;
    Print 'CDC Enabled for this database';
END
ELSE
BEGIN
    Print 'CDC is already enabled for this database';
END
GO
```

```
CDC Enabled for this database
```

Next, you enable CDC one table at a time. In this example, we'll apply it to the source table staging.SalesDetailSource. The incremental loading system we build around this CDC implementation will propagate all changes from the staging table to the fact table fact.SalesDetail.



```sql
--enable CDC on the staging table
EXEC sys.sp_cdc_enable_table
            @source_schema = 'staging',
            @source_name   = 'SalesDetailSource',
            @role_name     = null;

Print 'CDC Enabled on staging.SalesDetailSource.'
```

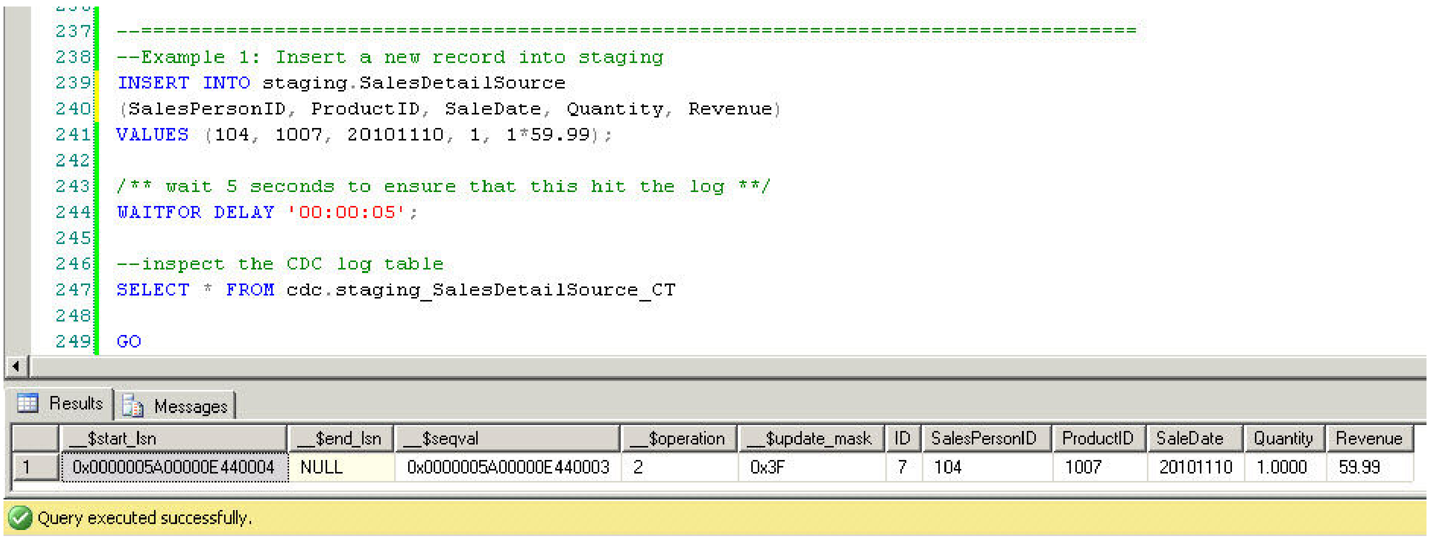
Messages

```
Job 'cdc.MMDB03_capture' started successfully.
Job 'cdc.MMDB03_cleanup' started successfully.
```

Query executed successfully.

The first time you enable a table for CDC in a given database, SQL Server creates the background SQL Agent jobs that listen for changes. Every table enabled for CDC automatically gets an identical table generated in the CDC system schema. This table has the same columns, plus a few extra to track the CDC mechanics. By default, this table is named **cdc.<schema name>_<table name>_CT**.

Now that CDC is enabled on the staging table, let's insert a record:



```sql
--==================================================================================
--Example 1: Insert a new record into staging
INSERT INTO staging.SalesDetailSource
(SalesPersonID, ProductID, SaleDate, Quantity, Revenue)
VALUES (104, 1007, 20101110, 1, 1*59.99);

/** wait 5 seconds to ensure that this hit the log **/
WAITFOR DELAY '00:00:05';

--inspect the CDC log table
SELECT * FROM cdc.staging_SalesDetailSource_CT

GO
```

Results | Messages

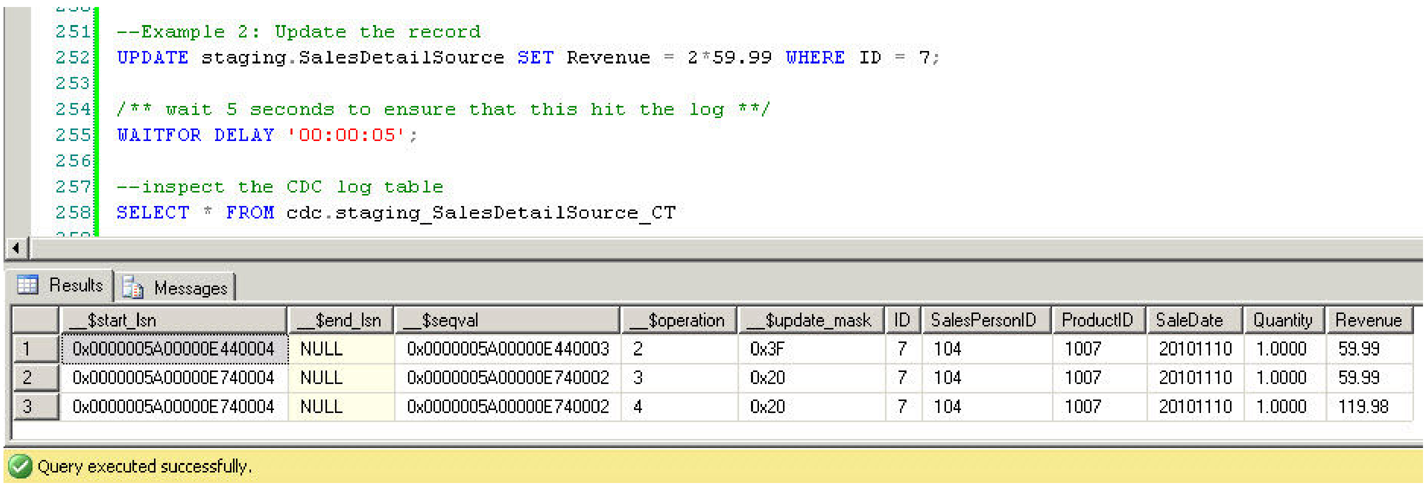
	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue
1	0x0000005A00000E440004	NULL	0x0000005A00000E440003	2	0x3F	7	104	1007	20101110	1.0000	59.99

Query executed successfully.

First, notice the 5-second delay built into this script. The delay is needed because the CDC tables are generally updated a few seconds later than the base tables, due to the asynchronous nature of the log reader. Thus, you must build a slight delay into your scripts when testing CDC.

Next, notice the extra columns added to the table called **__$start_lsn** and **__$operation**. SQL Server doesn't actually track the exact time of the update; instead, it uses the more accurate Log Sequence Number (LSN) in the __$start_lsn column. The __$operation column tracks the DML operation performed on the row (1:delete, 2:insert, 3:update before, 4:update after).

Now let's update a record:



```sql
--Example 2: Update the record
UPDATE staging.SalesDetailSource SET Revenue = 2*59.99 WHERE ID = 7;

/** wait 5 seconds to ensure that this hit the log **/
WAITFOR DELAY '00:00:05';

--inspect the CDC log table
SELECT * FROM cdc.staging_SalesDetailSource_CT
```

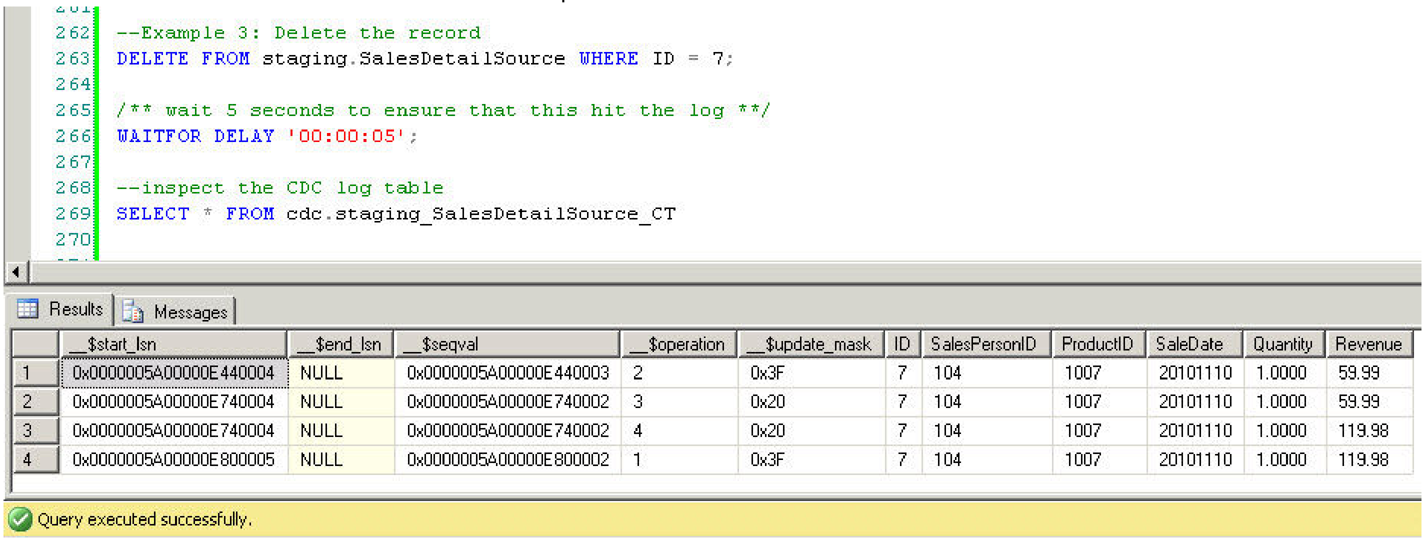
Results | Messages

	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue
1	0x0000005A00000E440004	NULL	0x0000005A00000E440003	2	0x3F	7	104	1007	20101110	1.0000	59.99
2	0x0000005A00000E740004	NULL	0x0000005A00000E740002	3	0x20	7	104	1007	20101110	1.0000	59.99
3	0x0000005A00000E740004	NULL	0x0000005A00000E740002	4	0x20	7	104	1007	20101110	1.0000	119.98

Query executed successfully.

For updates, notice that two change records are inserted: one with __$operation 3 (update before) and one with __$operation 4 (update after).

Finally, let's delete a record:



```
--Example 3: Delete the record
DELETE FROM staging.SalesDetailSource WHERE ID = 7;

/** wait 5 seconds to ensure that this hit the log **/
WAITFOR DELAY '00:00:05';

--inspect the CDC log table
SELECT * FROM cdc.staging_SalesDetailSource_CT
```

Results | Messages

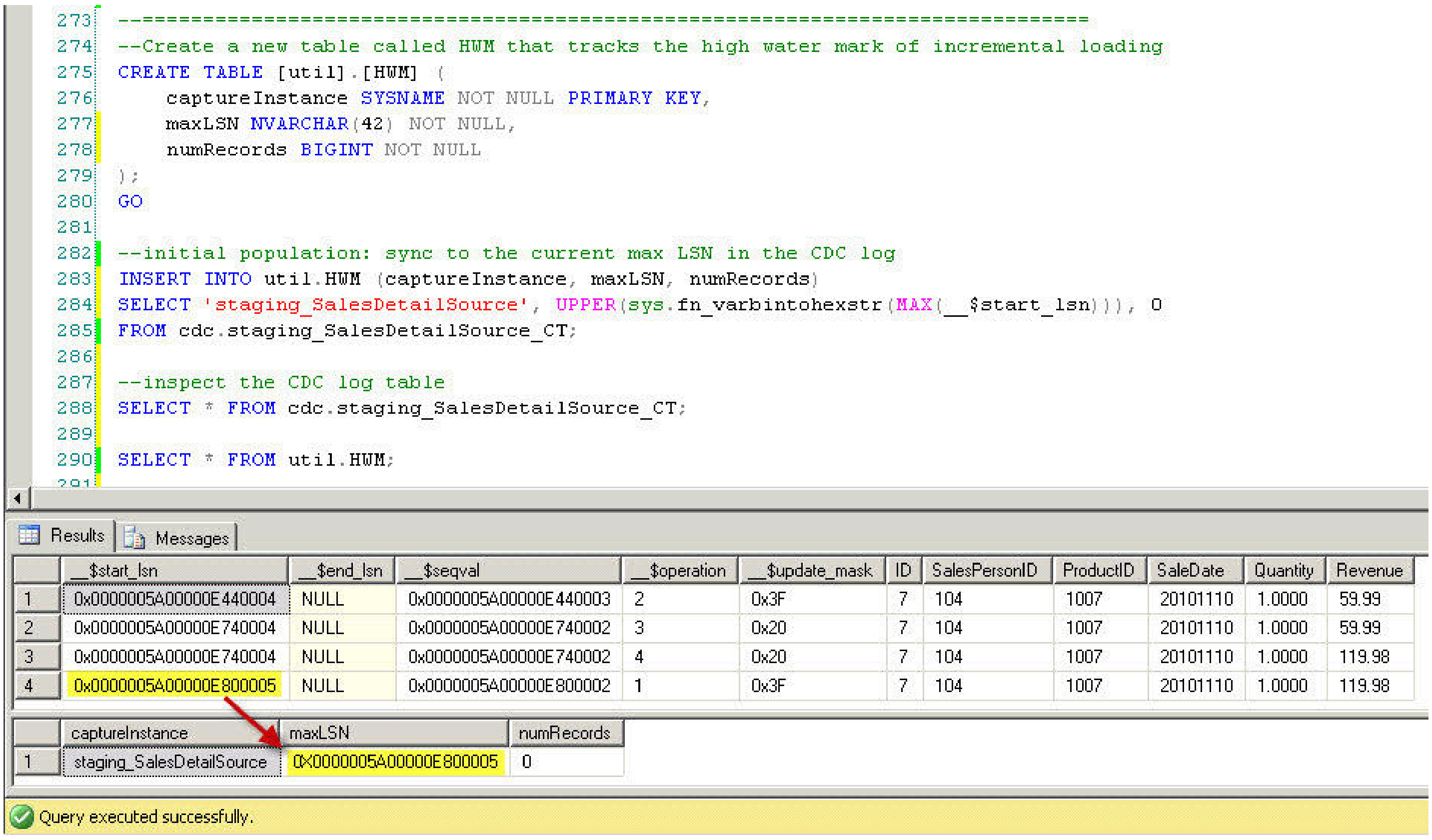
	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue
1	0x0000005A00000E440004	NULL	0x0000005A00000E440003	2	0x3F	7	104	1007	20101110	1.0000	59.99
2	0x0000005A00000E740004	NULL	0x0000005A00000E740002	3	0x20	7	104	1007	20101110	1.0000	59.99
3	0x0000005A00000E740004	NULL	0x0000005A00000E740002	4	0x20	7	104	1007	20101110	1.0000	119.98
4	0x0000005A00000E800005	NULL	0x0000005A00000E800002	1	0x3F	7	104	1007	20101110	1.0000	119.98

Query executed successfully.

For deletes, the __$operation is 1.

Incremental Load Framework

The keys to setting up an incremental load using CDC are to (1) source from the CDC log tables directly, and (2) keep track of how far each incremental load got, as tracked by the maximum LSN. First, I'll set up a tracking table called **util.HWM**, which stands for High Water Mark, as a tracking mechanism of how far all the latest successful incremental load has read. The util.HWM table has one row for each "capture instance." Note that more than one instance of CDC can be enabled for a table - as long as they have unique role names - so "capture instance" tracks the specific CDC instance on the table.



```
--================================================================================
--Create a new table called HWM that tracks the high water mark of incremental loading
CREATE TABLE [util].[HWM] (
    captureInstance SYSNAME NOT NULL PRIMARY KEY,
    maxLSN NVARCHAR(42) NOT NULL,
    numRecords BIGINT NOT NULL
);
GO

--initial population: sync to the current max LSN in the CDC log
INSERT INTO util.HWM (captureInstance, maxLSN, numRecords)
SELECT 'staging_SalesDetailSource', UPPER(sys.fn_varbintohexstr(MAX(__$start_lsn))), 0
FROM cdc.staging_SalesDetailSource_CT;

--inspect the CDC log table
SELECT * FROM cdc.staging_SalesDetailSource_CT;

SELECT * FROM util.HWM;
```

Results | Messages

	__$start_lsn	__$end_lsn	__$seqval	__$operation	__$update_mask	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue
1	0x0000005A00000E440004	NULL	0x0000005A00000E440003	2	0x3F	7	104	1007	20101110	1.0000	59.99
2	0x0000005A00000E740004	NULL	0x0000005A00000E740002	3	0x20	7	104	1007	20101110	1.0000	59.99
3	0x0000005A00000E740004	NULL	0x0000005A00000E740002	4	0x20	7	104	1007	20101110	1.0000	119.98
4	0x0000005A00000E800005	NULL	0x0000005A00000E800002	1	0x3F	7	104	1007	20101110	1.0000	119.98

	captureInstance	maxLSN	numRecords
1	staging_SalesDetailSource	0X0000005A00000E800005	0

Query executed successfully.

The INSERT statement initializes the HWM table to the current maximum LSN in the CDC log table. At this point, the staging and fact tables are in perfect synch.

Note the binary-to-hexadecimal conversion. LSNs are stored as BINARY(10) datatypes. If we ever wanted to port this routine to an SSIS package though, we would need to convert the LSNs to NVARCHAR, because SSIS doesn't support SQL BINARY datatypes. Thus, to maintain maximum code portability, I'm choosing to convert and store them all as NVARCHAR right from the start.

The conversions are:

```
--Convert from BINARY to NVARCHAR:
UPPER(sys.fn_varbintohexstr(<binary value>))

--Convert from NVARCHAR to BINARY:
```

```
CONVERT(BINARY(10), <nvarchar value>, 1)
```

Now, let's set up a view to query the staging table incrementally. The view needs to expose the __$start_lsn field, as we need to filter on this column when we query the view:

```
CREATE VIEW staging.vwSalesDetailCDC AS
 SELECT
 ------------------------------------------
 --CDC tracking column
 __$start_lsn
 ------------------------------------------
 --Dimension columns are selected as is
 ,ID
 ,SalesPersonID
 ,ProductID
 ,SaleDate
 ------------------------------------------
 --Numeric Facts get a multiplier
 --1 is delete, 2 is insert, 3 is update (before), 4 is update (after)
 ,CASE WHEN __$operation IN (1,3) then -1 ELSE 1 END * Quantity AS Quantity
 ,CASE WHEN __$operation IN (1,3) then -1 ELSE 1 END * Revenue AS Revenue
 FROM cdc.staging_SalesDetailSource_CT
GO
```

The key to the view is the method by which we expose the measure fields Quantity and Revenue. Any records inserted or updated need to be applied "as is" to the fact table, so we multiply the values by 1. Any records deleted or removed need to be negated from the fact table, so we need to multiply the values by -1. For example, if the revenue measure field is updated in the staging table from 100 to 110, the fact table will have one record with a value of -100 and one record with value 110, netting an increase of 10.

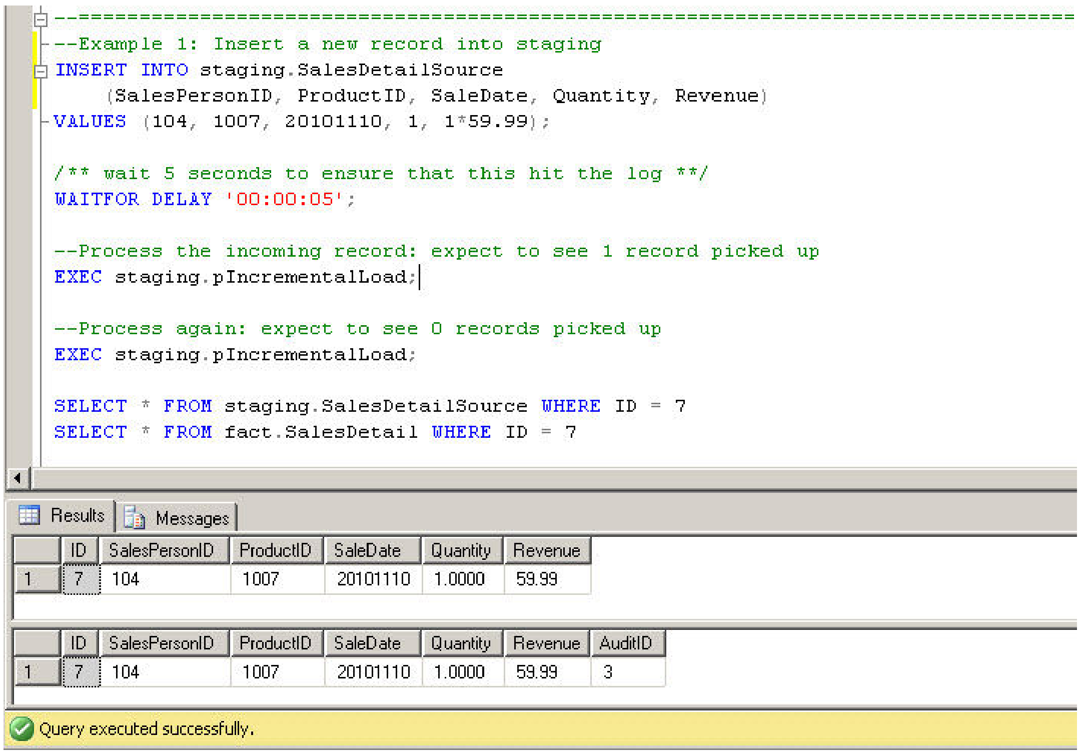
Next, let's set up a stored procedure that coordinates the incremental load:

```
CREATE PROCEDURE staging.pIncrementalLoad
AS
DECLARE @PreviousMaxLSN BINARY(10);
DECLARE @NewMaxLSN BINARY(10);
DECLARE @AuditID INT;
DECLARE @NumRecords INT;
DECLARE @CaptureInstance SYSNAME = 'staging_SalesDetailSource';
BEGIN TRAN

--Insert the new Audit record before loading data
INSERT INTO dim.Audit(CreateDate) VALUES (GETDATE());
SET @AuditID = SCOPE_IDENTITY();

--Get the previous high water mark LSN
SELECT @PreviousMaxLSN = CONVERT(BINARY(10), maxLSN, 1)
FROM util.HWM
WHERE captureInstance = @CaptureInstance;

--Get the new high water mark LSN
SELECT @NewMaxLSN = MAX(__$start_LSN) FROM staging.vwSalesDetailCDC;

--Exit the procedure if there are no updates to process
IF @NewMaxLSN IS NULL OR @NewMaxLSN = @PreviousMaxLSN
BEGIN
        COMMIT TRAN;
        Print 'Incremental load procedure returned without making any changes.'
        RETURN;
END;

--Insert all records since the previous max LSN, up to the new max LSN
INSERT INTO fact.SalesDetail (SalesPersonID, ProductID, SaleDate, Quantity, Revenue, AuditID)
SELECT SalesPersonID, ProductID, SaleDate, SUM(Quantity), SUM(Revenue), @AuditID
FROM staging.vwSalesDetailCDC
WHERE __$start_LSN > @PreviousMaxLSN
AND __$start_LSN <= @NewMaxLSN
GROUP BY SalesPersonID, ProductID, SaleDate;

SET @NumRecords = @@RowCount;

--Update the HWM table with the new LSN
UPDATE util.HWM
SET maxLSN = UPPER(sys.fn_varbintohexstr(@NewMaxLSN)),
        numRecords = @NumRecords
WHERE captureInstance = @CaptureInstance;

--Update the Audit record with information about this data load
UPDATE dim.Audit SET NumRecords = @NumRecords WHERE AuditID = @AuditID;

COMMIT TRAN

PRINT 'There were ' + CAST(@Numrecords AS VARCHAR) + ' records inserted in this incremental load.'
```

GO

The procedure queries the prior maximum High Water Mark, inserts the new increment of fact data, and then updates the new High Water Mark, all in the context of a transaction. Transactional consistency is critical in this procedure, as the updates to the fact table and the HWM table must either commit or roll back together. This will prevent any increments from getting skipped or doubled-up, as compared with the tracked HWM LSN.

Now, let's insert a new record into the staging table and run the incremental insert to replicate it to the staging table:



```
--================================================================================
--Example 1: Insert a new record into staging
INSERT INTO staging.SalesDetailSource
    (SalesPersonID, ProductID, SaleDate, Quantity, Revenue)
VALUES (104, 1007, 20101110, 1, 1*59.99);

/** wait 5 seconds to ensure that this hit the log **/
WAITFOR DELAY '00:00:05';

--Process the incoming record: expect to see 1 record picked up
EXEC staging.pIncrementalLoad;

--Process again: expect to see 0 records picked up
EXEC staging.pIncrementalLoad;

SELECT * FROM staging.SalesDetailSource WHERE ID = 7
SELECT * FROM fact.SalesDetail WHERE ID = 7
```

Results Messages

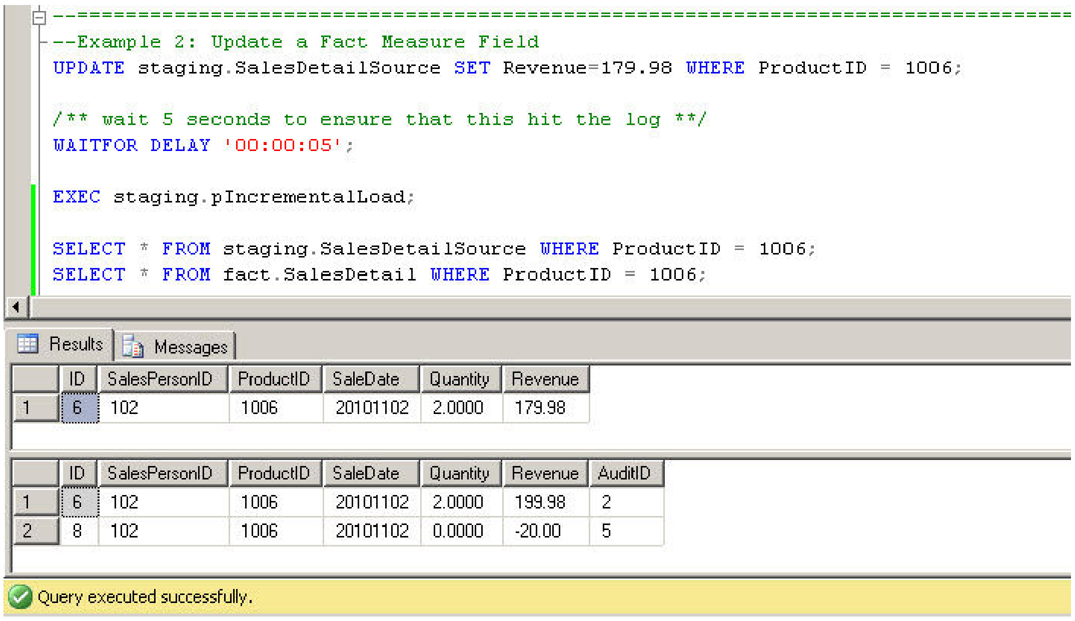
	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue
1	7	104	1007	20101110	1.0000	59.99

	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue	AuditID
1	7	104	1007	20101110	1.0000	59.99	3

Query executed successfully.

The first time the procedure runs it picks up the newly inserted record and updates the HWM table with the new maximum LSN. If you were to run the procedure again at this point, it would pick up 0 new records, as all changes have already been collected.

Next, let's run an update:



```
--================================================================================
--Example 2: Update a Fact Measure Field
UPDATE staging.SalesDetailSource SET Revenue=179.98 WHERE ProductID = 1006;

/** wait 5 seconds to ensure that this hit the log **/
WAITFOR DELAY '00:00:05';

EXEC staging.pIncrementalLoad;

SELECT * FROM staging.SalesDetailSource WHERE ProductID = 1006;
SELECT * FROM fact.SalesDetail WHERE ProductID = 1006;
```

Results Messages

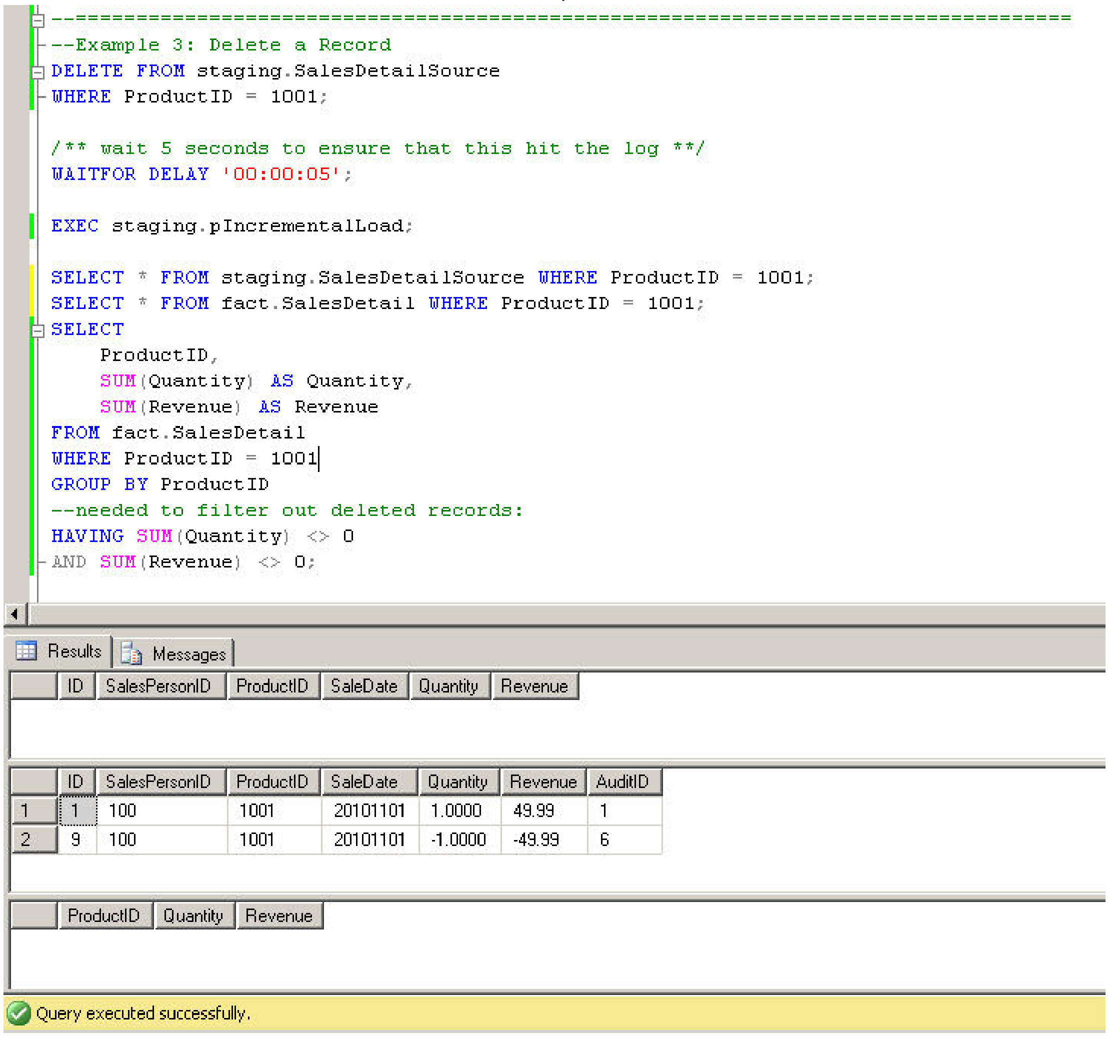
	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue
1	6	102	1006	20101102	2.0000	179.98

	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue	AuditID
1	6	102	1006	20101102	2.0000	199.98	2
2	8	102	1006	20101102	0.0000	-20.00	5

Query executed successfully.

As expected, a -20 revenue adjustment record is inserted with AuditID 5. Combined, they sum to the updated value of $179.98.

Finally, let's run a delete:



```
--==========================================================================================
--Example 3: Delete a Record
DELETE FROM staging.SalesDetailSource
WHERE ProductID = 1001;

/** wait 5 seconds to ensure that this hit the log **/
WAITFOR DELAY '00:00:05';

EXEC staging.pIncrementalLoad;

SELECT * FROM staging.SalesDetailSource WHERE ProductID = 1001;
SELECT * FROM fact.SalesDetail WHERE ProductID = 1001;
SELECT
    ProductID,
    SUM(Quantity) AS Quantity,
    SUM(Revenue) AS Revenue
FROM fact.SalesDetail
WHERE ProductID = 1001
GROUP BY ProductID
--needed to filter out deleted records:
HAVING SUM(Quantity) <> 0
AND SUM(Revenue) <> 0;
```

Results | Messages

	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue

	ID	SalesPersonID	ProductID	SaleDate	Quantity	Revenue	AuditID
1	1	100	1001	20101101	1.0000	49.99	1
2	9	100	1001	20101101	-1.0000	-49.99	6

	ProductID	Quantity	Revenue

Query executed successfully.

The record with ProductID 1001 is cancelled out of the fact table via an offsetting record. Note that any queries intending to filter out all deleted rows completely must include the HAVING clause as shown, to eliminate the cancellations.

Resynchronization

When implementing any incremental loading system, you must have scripts prepared that resynchronize data from the source to the destination to be run in the case of code upgrades, data anomalies, or other infrequent events that disrupt the system. The scripts must reload the destination tables and reset the HWM maximum LSN record -- but can't necessarily assume that the source system is quiet and not receiving updates while the script is running. This creates a timing challenge of getting the maximum LSN just right. One easy solution to this problem is to create a snapshot of the source database, then query the database snapshot to repopulate the destination data. The exact LSN of the database snapshot can be obtained by running the system-stored procedure **sys.sp_cdc_dbsnapshotLSN()**; this LSN can be stored in the HWM table to reset the precise starting point of the next incremental update.

Final Notes

CDC is a useful new feature in SQL Server 2008 Enterprise that allows developers to easily create incremental loading systems without using triggers or third-party products. The implementation shown here is modeled around querying its system tables directly. Other patterns for using CDC, as supported by the system-stored procedures and functions, are documented at MSDN. Whichever architecture you choose, I'm sure you'll find CDC a valuable addition to your ETL toolkit.