Migrating Data from Database to Database

by Carl Ganz, Jr.

As developers, most of work with at least three sets of data – production, test, and development. The problem with the latter two data sets is that the information in them becomes so mangled from our testing and development efforts that we soon lose any semblance of comparison to real data. Moreover, if we need to have users perform evaluation testing against these mangled data sets, they become frustrated when the data doesn’t looks like what they’ve been working with.

It’s very easy to backup and restore the production data over the test and development version. The problem here is that there are new tables, columns, views, and stored procedures which would be overwritten by doing so. This article will show a data-driven stored procedure that will copy the data from one server to another while leaving existing objects and structures intact.

The key idea is to determine which table/column combinations exist on both the source and the target, and then migrate the data in just those tables and columns. There are a number of issues that must be dealt with when migrating data in this fashion, especially when foreign key constraints enter the picture. First, you need to turn off the table constraints. This does not remove the constraints. It simply ignores them until they are tuned back on so as to avoid any further unintended consequences. Likewise, if you have any triggers on your tables, you may wish to disable these as well. The final goal is to create a list of the SQL commands that will effect the migration. These commands will be added to a temp table and then executed in sequence.

First, you’ll need to decide if any of the tables should not be migrated. For example, user permissions tables would likely not be appropriate as your permissions on the production database will be far less than those available for you on development. You can prepare a list of these table by creating a table variable and populating it as shown in Listing 1.

Listing 1 – Excluding tables from migration

DECLARE @SkipTables TABLE

(

TableName varchar(100)

)

INSERT INTO @SkipTables (TableName) VALUES ('User\_Permission')

Next, you can pull a list of all the constraints from the INFORMATION\_SCHEMA view and suspend them by applying the NOCHECK option to them. You can accomplish this via the code in Listing 2.

Listing 2 – Turn off the constraints

INSERT INTO #SQLtemp

SELECT 'ALTER TABLE [' + SCHEMA\_NAME(schema\_id) + '].' + OBJECT\_NAME(parent\_object\_id) +

' NOCHECK CONSTRAINT ' + OBJECT\_NAME(OBJECT\_ID)

FROM sys.objects

WHERE type\_desc LIKE '%CONSTRAINT'

Then, you’ll need to delete the data in the target tables. Unfortunately, we can’t use the TRUNCATE statement here as these tables may have foreign key constraints and TRUNCATE will not work in these cases. You’ll need to perform the much slower process of DELETEing the data. These commands can be created via the SQL in Listing 3.

Listing 3 – Deleting the data in the target tables

INSERT INTO #SQLtemp

SELECT 'DELETE FROM ' + t1.TABLE\_SCHEMA + '.[' + t1.TABLE\_NAME + ']'

FROM INFORMATION\_SCHEMA.tables t1

INNER JOIN [EINSTEIN].[Contracts].INFORMATION\_SCHEMA.tables t2 ON t1.TABLE\_NAME = t2.TABLE\_NAME

WHERE t1.TABLE\_TYPE = 'BASE TABLE'

AND t1.TABLE\_NAME NOT IN (SELECT TABLE\_NAME

FROM INFORMATION\_SCHEMA.columns

WHERE DATA\_TYPE = 'xml')

AND t1.TABLE\_NAME NOT IN (SELECT TableName FROM @SkipTables)

ORDER BY t1.TABLE\_NAME

We only wish to DELETE data from those tables that exist in both the source and target databases. Those tables that do not exist in both are assumed to be works in progress and are left alone. Another major consideration here is the issue of referential integrity. If there is, say, a Dictionary table that contains foreign key references to a series of other tables, these other tables must be deleted first before the Dictionary table is deleted. Otherwise, a referential integrity error will occur. One way to avoid this is to use the extended properties of the table objects to establish a sort order value so such objects as Dictionary tables are cleaned out last.

*Note: The reason tables with XML columns are excluded is due to the inability of such tables to participate in distributed queries. Even pulling only the non-XML columns is not permitted. Such tables can be handled by creating views that pull all but the XML columns. By extracting the data from these views you can successfully perform the migration.*

Since your database may be quite large, you should speak with your DBA about allocating the required space or turning off the logging that accompanies such massive data deletions and inserts. You could easily exceed the allocated space for the database and your migration will end in the middle. A data migration is normally far too large to wrap a transaction around for a rollback so you’ll be left with sections of your data missing. Though this example does not show it, you may wish to use SQL Server’s email features to send an email should the migration terminate before its completes.

Once the data has been cleaned out of the target, the next step is to create a temp table of those tables and column combinations that exist in both databases. The code in Listing 4 will perform this feat.

Listing 4 – Create a list of table/column names

INSERT INTO #Tabletemp

SELECT c1.TABLE\_SCHEMA, c1.TABLE\_NAME, c1.COLUMN\_NAME

FROM [Contracts].INFORMATION\_SCHEMA.columns c1

INNER JOIN INFORMATION\_SCHEMA.tables t1 ON c1.TABLE\_NAME = t1.TABLE\_NAME

INNER JOIN [EINSTEIN].[Contracts].INFORMATION\_SCHEMA.columns c2 ON c1.TABLE\_NAME + c1.COLUMN\_NAME = c2.TABLE\_NAME + c2.COLUMN\_NAME

WHERE t1.TABLE\_TYPE = 'BASE TABLE'

AND c1.TABLE\_NAME NOT IN (SELECT TABLE\_NAME

FROM INFORMATION\_SCHEMA.columns

WHERE DATA\_TYPE = 'xml')

AND c1.TABLE\_NAME NOT IN (SELECT TableName FROM @SkipTables)

ORDER BY c1.TABLE\_NAME, c1.ORDINAL\_POSITION

With this in place, you can iterate through each column in a given table to create a SQL statement that looks like this:

INSERT INTO Employees (EmployeeID, LastName, FirstName)

SELECT EmployeeID, LastName, FirstName

FROM [SourceServer].[MyDB]. Employees WITH (NOLOCK)

If this table has an IDENTITY key, the INSERT..SELECT statement will be preceded by SET IDENTITY\_INSERT Employees ON and followed by SET IDENTITY\_INSERT Employees OFF. This will allow migration of the primary key column. Here, as well, the INSERT…SELECTs must be performed in a certain order so as to avoid referential integrity conflicts. Tables like the Dictionary example with all the foreign key pointers need to be populated first this time.

In the final step, all the constraints need to be turned back on. Now that we have a temp table filled will all the SQL statements to perform a data migration, we can execute these SQL statements in sequence. The code in Listing 5 will iterate the SQL commands in the temp table and execute them one by one.

Listing 5 – Execute each SQL statement in turn

SELECT @Cnt = MAX(ID) FROM #SQLtemp

SET @x = 1

WHILE @x <= @Cnt

BEGIN

SELECT @SQL = SQL

FROM #SQLtemp

WHERE ID = @x

BEGIN TRY

SET @StartTime = GETDATE()

EXEC(@SQL)

SET @ElapsedTime = DATEDIFF(SECOND, @StartTime, GETDATE())

--Write every successfully executed SQL command to SyncLog

INSERT INTO SyncLog

(ErrorNumber, Message, SQL, ErrorDate, ElapsedTime)

VALUES

(Null, 'OK', @SQL, GETDATE(), @ElapsedTime)

END TRY

BEGIN CATCH

SET @SQLError = @@ERROR

--If an error was found, write it to the SyncLog table.

--One of the most common errors will be caused by trying to insert a value from a larger column

--into that of a smaller column. This will happen if you reduced the size of a column in

--your target to less than that of your source. In other cases, the data type may have changed

--and this will throw an error as well.

IF @SQLError <> 0

INSERT INTO SyncLog

(ErrorNumber, Message, SQL, ErrorDate)

VALUES

(@SQLError, Error\_Message(), @SQL, GETDATE())

END CATCH

SET @x = @x + 1

END

For each table, the stored procedure will execute SQL statements similar to those shown in Listing 6.

Listing 6 – SQL statements to migrate data

ALTER TABLE [dbo].MyTable NOCHECK CONSTRAINT PK\_MyConstraint

DELETE FROM dbo.[MyTable]

SET IDENTITY\_INSERT dbo.[MyTable] ON (if applicable)

INSERT INTO dbo.[MyTable] ([MyColumn1],[MyColumn2]) SELECT [MyColumn1],[MyColumn2] FROM [MyServer].[MyDataBase].dbo.[MyColumn1]

SET IDENTITY\_INSERT dbo.[MyTable] OFF (if applicable)

ALTER TABLE [dbo].MyTable CHECK CONSTRAINT PK\_MyConstraint

Because this stored procedure is completely data-driven, it will work with any SQL Server database. The only customization required is altering the name of the source server and database and enumerating the tables you wish to exclude.

ALTER PROCEDURE [dbo].[spc\_CopyProd2Test]

AS

/\*

This stored procedure pulls the contents of a database on one server into the table structures of a database on another server. You can change the source server.database by doing a global replace of this string:

[MYSERVER].[Contracts]

with the appropriate servername.databasename

The commands it issues for each table are at follows:

ALTER TABLE [dbo].MyTable NOCHECK CONSTRAINT PK\_MyConstraint

DELETE FROM dbo.[MyTable]

SET IDENTITY\_INSERT dbo.[MyTable] ON (if applicable)

INSERT INTO dbo.[MyTable] ([MyColumn1],[MyColumn2]) SELECT [MyColumn1],[MyColumn2] FROM [MyServer].[MyDataBase].dbo.[MyColumn1]

SET IDENTITY\_INSERT dbo.[MyTable] OFF (if applicable)

ALTER TABLE [dbo].MyTable CHECK CONSTRAINT PK\_MyConstraint

This procedure will remove all constraints, SET IDENTITY\_INSERT ON for all tables which have identity

columns, delete the contents of the table, INSERT..SELECT the data from the source table to the target,

SET IDENTITY\_INSERT OFF as appropriate, and re-enable the constraints.

\*/

DECLARE @SQL varchar(max)

DECLARE @SQLTemplate varchar(max)

DECLARE @Columns varchar(max)

DECLARE @SQLError varchar(max)

DECLARE @StartTime datetime

DECLARE @ElapsedTime int

DECLARE @TableSchemaOrig varchar(100)

DECLARE @TableSchema varchar(100)

DECLARE @TableNameOrig varchar(100)

DECLARE @TableName varchar(100)

DECLARE @ColumnName varchar(100)

DECLARE @HasIdentityKey bit

DECLARE @Cnt int

DECLARE @x int

--If there are certain tables on the target you wish left alone,

--INSERT their names into the @SkipTables table variable

DECLARE @SkipTables TABLE

(

TableName varchar(100)

)

INSERT INTO @SkipTables (TableName) VALUES ('SyncLog')

INSERT INTO @SkipTables (TableName) VALUES ('User\_Permission')

INSERT INTO @SkipTables (TableName) VALUES ('User\_X\_Permission')

INSERT INTO @SkipTables (TableName) VALUES ('SystemState')

INSERT INTO @SkipTables (TableName) VALUES ('IncomingFax')

INSERT INTO @SkipTables (TableName) VALUES ('IncomingFaxPages')

INSERT INTO @SkipTables (TableName) VALUES ('ContractConfig')

--Create a table to store error and progress messages

IF NOT EXISTS (SELECT 1 FROM dbo.sysobjects WHERE id = object\_id('SyncLog'))

BEGIN

CREATE TABLE SyncLog

(

ErrorNumber int,

Message varchar(max),

SQL varchar(max),

ErrorDate datetime,

ElapsedTime int

)

END

ELSE

TRUNCATE TABLE SyncLog

--Indicate when the process has begun

INSERT INTO SyncLog

(ErrorNumber, Message, SQL, ErrorDate)

VALUES

(Null, 'Migration begun', Null, GETDATE())

--This will hold the SQL statements that are created

CREATE TABLE #SQLtemp

(

ID int identity(1,1),

SQL varchar(max)

)

--This will hold the structures of the tables to create comma-separated column lists

CREATE TABLE #Tabletemp

(

ID int identity(1,1),

TABLE\_SCHEMA varchar(100),

TABLE\_NAME varchar(100),

COLUMN\_NAME varchar(100)

)

--Create SQL to turn off all the constraints

INSERT INTO #SQLtemp

SELECT 'ALTER TABLE [' + SCHEMA\_NAME(schema\_id) + '].' + OBJECT\_NAME(parent\_object\_id) +

' NOCHECK CONSTRAINT ' + OBJECT\_NAME(OBJECT\_ID)

FROM sys.objects

WHERE type\_desc LIKE '%CONSTRAINT'

--Create SQL to truncate the tables for those tables which exist in both the source and target databases

INSERT INTO #SQLtemp

SELECT 'DELETE FROM ' + t1.TABLE\_SCHEMA + '.[' + t1.TABLE\_NAME + ']'

FROM INFORMATION\_SCHEMA.tables t1

INNER JOIN [EINSTEIN].[Contracts].INFORMATION\_SCHEMA.tables t2 ON t1.TABLE\_NAME = t2.TABLE\_NAME

WHERE t1.TABLE\_TYPE = 'BASE TABLE'

AND t1.TABLE\_NAME NOT IN (SELECT TABLE\_NAME

FROM INFORMATION\_SCHEMA.columns

WHERE DATA\_TYPE = 'xml')

AND t1.TABLE\_NAME NOT IN (SELECT TableName FROM @SkipTables)

ORDER BY t1.TABLE\_NAME

--Create SQL to insert data for those tables and columns which exist in both the source and target databases

INSERT INTO #Tabletemp

SELECT c1.TABLE\_SCHEMA, c1.TABLE\_NAME, c1.COLUMN\_NAME

FROM [Contracts].INFORMATION\_SCHEMA.columns c1

INNER JOIN INFORMATION\_SCHEMA.tables t1 ON c1.TABLE\_NAME = t1.TABLE\_NAME

INNER JOIN [EINSTEIN].[Contracts].INFORMATION\_SCHEMA.columns c2 ON c1.TABLE\_NAME + c1.COLUMN\_NAME = c2.TABLE\_NAME + c2.COLUMN\_NAME

WHERE t1.TABLE\_TYPE = 'BASE TABLE'

AND c1.TABLE\_NAME NOT IN (SELECT TABLE\_NAME

FROM INFORMATION\_SCHEMA.columns

WHERE DATA\_TYPE = 'xml')

AND c1.TABLE\_NAME NOT IN (SELECT TableName FROM @SkipTables)

ORDER BY c1.TABLE\_NAME, c1.ORDINAL\_POSITION

SET @Cnt = @@ROWCOUNT

SET @x = 1

SET @HasIdentityKey = 0

SET @TableSchemaOrig = ''

SET @TableNameOrig = ''

SET @Columns = ''

--This is the SQL template for the INSERT...SELECT. The strings delimited by percent signs will be

--replaced with actual values

SET @SQLTemplate = 'INSERT INTO %Table% (%Columns%) SELECT %Columns% FROM [EINSTEIN].[Contracts].%Table% WITH (NOLOCK)'

SET @SQL = @SQLTemplate

--For each table/column entry in #Tabletemp

WHILE @x <= @Cnt

BEGIN

--Pull the information for a table/column entry

SELECT @TableSchema = TABLE\_SCHEMA,

@TableName = TABLE\_NAME,

@ColumnName = COLUMN\_NAME

FROM #Tabletemp

WHERE ID = @x

--If this is the first time through then assign the values of

--the starting schema and table name

IF @TableSchemaOrig = ''

SET @TableSchemaOrig = @TableSchema

IF @TableNameOrig = ''

SET @TableNameOrig = @TableName

--If we haven't moved to a new table then keep building the column list

IF @TableNameOrig = @TableName

SET @Columns = @Columns + '[' + @ColumnName + '],'

--When all the table's column have been retrieved create SET IDENTITY\_INSERT ON/OFF

--statements for those tables that have identity columns

IF @TableNameOrig <> @TableName

BEGIN

--The SET IDENTITY\_INSERT <tablename> ON/OFF is appended to the INSERT..SELECT SQL

--because executing them individually was causing problems. Therefore, the commands are

--executed as a semi-colon separated batch like this:

--SET IDENTITY\_INSERT dbo.[MyTable] ON (if applicable);

--INSERT INTO dbo.[MyTable] ([MyColumn1],[MyColumn2]) SELECT [MyColumn1],[MyColumn2] FROM [MyServer].[MyDataBase].dbo.[MyColumn1];

--SET IDENTITY\_INSERT dbo.[MyTable] OFF (if applicable)

IF EXISTS (SELECT TABLE\_NAME

FROM INFORMATION\_SCHEMA.COLUMNS

WHERE TABLE\_SCHEMA = @TableSchemaOrig

AND TABLE\_NAME = @TableNameOrig

AND COLUMNPROPERTY(object\_id(TABLE\_NAME), COLUMN\_NAME, 'IsIdentity') = 1)

BEGIN

SET @HasIdentityKey = 1

END

--Remove the trailing comma

SET @Columns = SUBSTRING(@Columns, 1, LEN(@Columns) - 1)

--Replace the placeholders with the table name and comma-delimited field list

SET @SQL = REPLACE(@SQL, '%Table%', @TableSchema + '.[' + @TableNameOrig + ']')

SET @SQL = REPLACE(@SQL, '%Columns%', @Columns)

--Create the SET IDENTITY\_INSERT <tablename> OFF command if needed

IF @HasIdentityKey = 1

BEGIN

SET @SQL = 'SET IDENTITY\_INSERT ' + @TableSchemaOrig + '.[' + @TableNameOrig + '] ON;' + @SQL

SET @SQL = @SQL + ';SET IDENTITY\_INSERT ' + @TableSchemaOrig + '.[' + @TableNameOrig + '] OFF'

END

INSERT INTO #SQLtemp (SQL) VALUES (@SQL)

SET @SQL = @SQLTemplate

SET @Columns = ''

SET @TableSchemaOrig = @TableSchema

SET @TableNameOrig = @TableName

SET @HasIdentityKey = 0

END

ELSE

SET @x = @x + 1

END

DROP TABLE #Tabletemp

--Create the SQL to re-enable the constraints

INSERT INTO #SQLtemp

SELECT 'ALTER TABLE [' + SCHEMA\_NAME(schema\_id) + '].' + OBJECT\_NAME(parent\_object\_id) +

' CHECK CONSTRAINT ' + OBJECT\_NAME(OBJECT\_ID)

FROM sys.objects

WHERE type\_desc LIKE '%CONSTRAINT'

--Now that the SQL statements needed for the migration are ready and in

--the required order, execute each one

SELECT @Cnt = MAX(ID) FROM #SQLtemp

SET @x = 1

WHILE @x <= @Cnt

BEGIN

SELECT @SQL = SQL

FROM #SQLtemp

WHERE ID = @x

BEGIN TRY

SET @StartTime = GETDATE()

EXEC(@SQL)

SET @ElapsedTime = DATEDIFF(SECOND, @StartTime, GETDATE())

--Write every successfully executed SQL command to SyncLog

INSERT INTO SyncLog

(ErrorNumber, Message, SQL, ErrorDate, ElapsedTime)

VALUES

(Null, 'OK', @SQL, GETDATE(), @ElapsedTime)

END TRY

BEGIN CATCH

SET @SQLError = @@ERROR

--If an error was found, write it to the SyncLog table.

--One of the most common errors will be caused by trying to insert a value from a larger column

--into that of a smaller column. This will happen if you reduced the size of a column in

--your target to less than that of your source. In other cases, the data type may have changed

--and this will throw an error as well.

IF @SQLError <> 0

INSERT INTO SyncLog

(ErrorNumber, Message, SQL, ErrorDate)

VALUES

(@SQLError, Error\_Message(), @SQL, GETDATE())

END CATCH

SET @x = @x + 1

END

--Indicate when the process has ended

INSERT INTO SyncLog

(ErrorNumber, Message, SQL, ErrorDate)

VALUES

(Null, 'Migration completed', Null, GETDATE())

--SELECT \* FROM #SQLtemp ORDER BY ID

DROP TABLE #SQLtemp