Curious Developments

BI Builder:

An Open-Source T-SQL Application for Microsoft SQL Server Data Warehouse Developers

Version alpha-0.1.0

Developer Guide

By Pat Connors, Curious Developments

Executive Summary of BI Builder

BIB is an open-source T-SQL application for MSSQL data warehouse developers: It generates staging and dimension tables as well as ETL stored procedures that manage data movements between them.

Designed for ease of use and customisation, BIB has one stored procedure that you run to build your database layers. BI Builder is composed of a series of modular stored procedures and scalar functions that you can modify to suit your style of building data warehouses.

BI Builder has been tested on MSSQL versions 2008R2, 2012 and 2014 CTP.

Open Source Licence And Copyright

Contents

BI Builder Overview

BIB is an open-source T-SQL application for MSSQL data warehouse developers: It generates staging and dimension tables as well as ETL stored procedures that manage data movements between them.

Designed for ease of use and customisation, BIB has one stored procedure that you run to build your database layers. BI Builder is composed of a series of modular stored procedures and scalar functions that you can modify to suit your style of building data warehouses.

BI Builder Features

BIB generates objects and SQL that accounts for many considerations you have as a data warehouse developer:

- BIB produces both Type 1 and Type 2 Slowly Changing Dimensions (SCD) – all you need to do to create a Type 2 SCD is supply to the stored procedure [bib].[Main] a list of the fields that will trigger creation of a new active dimension record.

- BIB stores mappings between columns in your staging table and related dimension tables as well as metadata about the columns and your design decisions about them.

- BIB builds an ETL system between staging and dimensions that can properly handle records that are either new, updated, stale or deleted.

- The generated ETL SQL loads data from staging to dimensions in batches of customisable size, letting you tune your loads to handle massive updates with maximum efficiency.

- BIB handles Schemas and Filegroups with ease, allowing you to optimise how you structure the physical layout of your data warehouse.

- BIB automatically detects columns containing information about when the record was last modified allowing the generated ETL stored procedures to protect your data from stale overwrites.

- BIB automatically detects columns that flag whether or not a record has been soft-deleted, generating an ETL that handles deletion of records from dimensions if that suits your requirements.

- BIB generates dimensions and ETL stored procedures for really large tables with as much ease as small table – see the example below in this document where the “Model Table” has 300 fields.

- BIB handles composite primary keys automatically.

- BIB generates ETL stored procedures that stores the lineage of the data and logs all data movements (Inserts, Updates and Deletions) as well as any error encountered during data loads.

- BIB integrates with AutoAudit and applying auditing to your generated dimensions is as easy as setting a parameter when you run the ETL build process.

- BIB applies unique indexes based on business keys (*primary keys in the source data*) that protect your dimension table from the import of duplicate records – should records be added by means other than the generated ETL stored procedures.

- BIB handles surrogate keys in a standardised manner and adds standardised columns to both staging and dimension tables that let you determine the state of data and loads as well as lock data from ever being updated during an ETL cycle.

- BIB handles the generation and maintenance of default records – a record with a surrogate dimension key of "-1" that is used when a fact table has a NULL value instead of a key into the dimension.

- BIB adapts any check constraints you supply with your "Model Table" through to the Dimension, allowing another level of data quality assurance.

- BIB can automatically apply type conversions, such as when date values in your source system are expressed as integers and you want to convert them to MSSQL DateTime. There is an example of this described below.

- BIB can automatically exclude staging table columns from featuring in the dimension and ETL and you can easily add or remove rules for exclusion.

- BIB can automatically run data analysis across the sample data you provide in the "Model Table" – detecting, for example, if the data type is not optimal for the values contained within the column. It will also tell you what percentage of the values are NULL and how many distinct values are held within each column. That last feature can be useful when designing hierarchies in Analysis Services.

- BIB can automatically make column names more readable from a business perspective when you have loaded your dimensions in to Analysis Services. See the Readability example below.

- BIB produces useful scripts you can use to test the generated objects and stored procedures.

- BIB lives solely in your development database – there are no dependencies on the [BIB] schema for the objects you will deploy to your testing and production data warehouses.

- BIB is easily modified to produce tables and SQL that better suit your development goals.

Learning BI Builder using examples

Supplied are a series of examples demonstrating what BI Builder does and how to drive it to build the staging and dimension layers of your data warehouse.

The first three steps below are exactly the same for installing BI Builder into either an existing or empty database to use in data warehouse development. The fourth step only installs the examples.

Step 1 – Create a new empty database in SQL Server Management Studio

Using Microsoft's SQL Server Management Studio (SMSS), create a new database named "BIB_Examples" (or any name you want) and specify "Simple" as the recovery model.

Step 2 –Optionally apply AutoAudit to the examples database

AutoAudit is an excellent MSSQL auditing library that generates triggers to track changes to both data and schema. It is wholly written in T-SQL and is simple to install on new and existing databases.

BI Builder works fine without it but using AutoAudit or equivalent audit function enabled in your data warehouse in order to give you greater insight into changes

If you would like to have AutoAudit installed, download the script from http://AutoAudit.codeplex.com and execute it in a query window against the "BIB_Examples" database.

The version used alongside BI Builder during development was version 3.30a.

```
AutoAudit script for SQL Server 2005, 2008, 2008R2, 2012
(c) 2007-2013 Paul Nielsen Consulting, inc.
www.sqlserverbible.com
AutoAudit.codeplex.com
Created by Paul Nielsen
Coded by Paul Nielsen and John Sigouin
```

The author's preference is to modify the AutoAudit setup script so audit information ends up in its own FILEGROUP for ease of management.

Step 3 – Install BI Builder Setup Script

In a new query window in SMSS, load and execute the script named:

```
BI Builder <version> Development Server Setup Script.sql
```

(Current version is alpha-0.1.0)

Step 4 – Install the BI Builder Examples Setup Script

In a new query window in SMSS, load and execute the script named:

```
BI Builder <version> Examples Setup Script.sql
```

The example database is now ready for use

Working through the Examples

The prepared examples show what BI Builder can produce for you without any customisation

Customisation of BI Builder is covered later in this documentation.

In each example, you will run a single stored procedure that you provide with parameters. This procedure is **[bib].[Main]**.

The first example, building a dimension and ETL stored procedures for a simple table, will go into detail as to what is created. The subsequent examples will be brief.

Terms used in the BI Builder examples

The term ***Model Table*** refers to a table you have placed in your development database as a starting model of a **Staging Table**. It has all the columns you will be importing in the future as data updates from your source system(s). This table needs to have a primary key defined so BI Builder can determine the business keys to use in the ETL logic.

Ideally this table will be populated with a good sample of the data you will be feeding into your data warehouse but that is not essential in order to build the ETL with BI Builder.

The ***Model Table*** is only used in the BI Builder ETL development process – firstly as a model and then as a repository of data used in testing the generated ETL. It does not go into production.

The term ***Staging Table*** refers to a table that BI Builder will create from your ***Model Table***. It is essentially your ***Model Table*** augmented with a new surrogate key and additional fields to manage the ETL process.

In production, your external systems will deposit data into the generated ***Staging Table*** so that it will then be fed into the generated data warehouse ***Dimension Table*** by the generated ETL stored procedures.

The term ***ETL*** is used here as shorthand, and in a limited sense, to mean “controlled and logged movements of data between its deposition into the data warehouse staging layer and its ingestion into the dimensional layer”.

Metadata refers to information about the generated tables that BI Builder uses to construct them and related stored procedures. BI Builder gives you the facility to pause a build after metadata has been collected, make some edits to the metadata and then resume the build. Why you might benefit from that and how is described in Example 11.

Example 1: A Simple Type 1 Dimension

This example uses a simple table to demonstrate clearly what gets produced.

Load the SQL file "**`Example 1 A Simple Type 1 Dimension.sql`**" into a query window.

```
-- Example 1 A Simple Type 1 Dimension

-- ETL BUILD SCRIPT FOR TABLE [Models].[Tractors]
-----------------------------------------------------------------------------

DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @SourceFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
   SET @ModelSchemaName = 'Models'
   SET @ModelTableName = 'Tractors'

-- Staging Table Details ------------
   SET @SourceFilegroupName = 'Example_Staging'
   SET @StagingSchemaName = 'Example_Staging'
   SET @StagingTableName = 'Tractors'
   SET @SlowlyChangingDimensionColumnList = NULL

-- Dimension Table Details ------------
   SET @DimSchemaName = 'Examples'
   SET @DimTableName = 'DimTractors'
   SET @DimFilegroupName  = 'Examples'
   SET @SourceSystem_Id  = 1

-- Control Flags for BI Builder Main()
   SET @SkipDataAnalysis = 0
   SET @ApplyAutoAudit = 0

   ---- 1) Control Flags for automated end-to-end build process
   SET @PauseForManualEditsToMetadata = 0
   SET @UseCurrentMetadataForBuild = 0

EXECUTE[bib].[Main]
    @ModelSchemaName
   ,@ModelTableName
   ,@StagingSchemaName
   ,@StagingTableName
   ,@SourceFilegroupName
   ,@SourceSystem_Id
   ,@SlowlyChangingDimensionColumnList
   ,@DimSchemaName
   ,@DimTableName
   ,@DimFilegroupName
   ,@SkipDataAnalysis
   ,@PauseForManualEditsToMetadata
   ,@UseCurrentMetadataForBuild
   ,@ApplyAutoAudit

GO
```

This table is the model - ***[Models].[Tractors]*** - from which the staging and dimension tables will be built as well as the ETL stored procedures.

The generated staging table will be called ***[Example_Staging].[Tractors]*** and will be created on a new filegroup called ***Example_Staging***

The generated dimension table will be called ***[Examples].[DimTractors]*** and will be created on a new filegroup called ***Examples***

For this example, we will be doing a straight-through build so we don't need to set these control flags

Handy Tip:

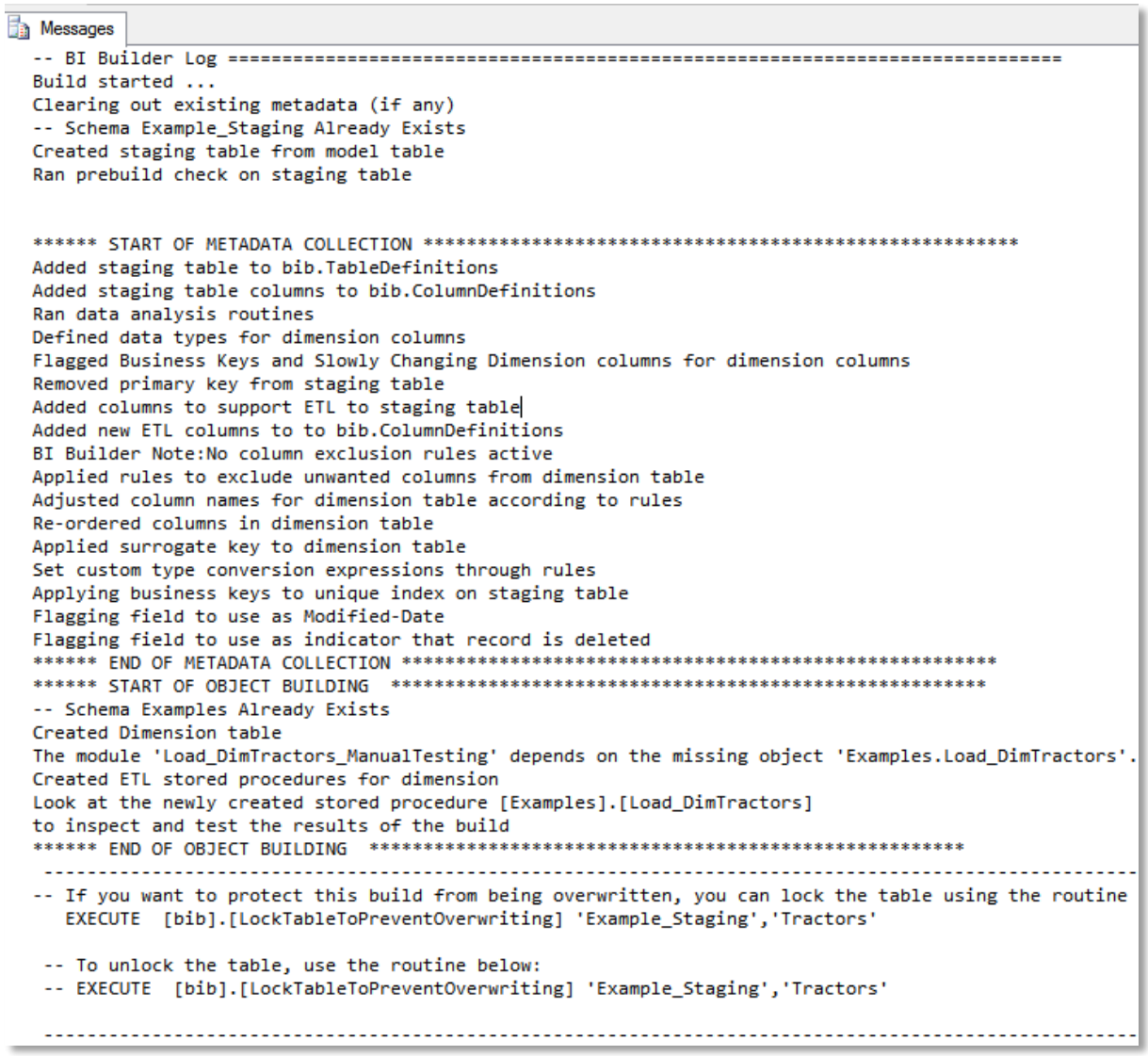
You can create the script shown above using a helper routine in an SMSS query window:

```
EXECUTE [BIB].[Main_PrintScriptToBuildETLFromModelTable]
                @ModelSchemaName = 'Models'
              , @ModelTableName  = 'Tractors'
```

Now execute the query and we will examine the results.

The first thing you see is the BI Builder message log in the Messages window where it lists the steps it has undertaken.

BI Builder will also put warnings into this log and SQL statements you can use to address any problems it has found. We will have examples of that below and will describe how to pause builds in order to edit the metadata used to build the ETL.



This example has created the following objects:

1) The staging table **[Example_Staging].[Tractors]** and populated it with the data held in the model table **[Models].[Tractors]** so it is ready for testing with.

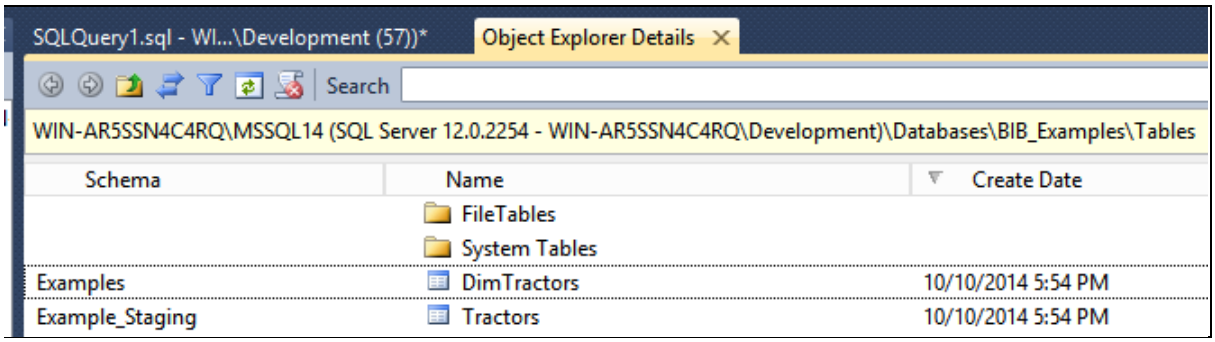
2) The dimension table **[Examples].[DimTractors]**

3) A main stored procedure that drives the loading of data into this dimension - **[Examples].[Load_DimTractors]**. It will be examined in detail below.

4) A stored procedure that is called by the main procedure and ensures that there is a default record in the dimension - **[Examples].[Load_DimTractors_SetDefaultValues]**

5) A stored procedure that is called by the main procedure that flags records in the staging table to be loaded - **[Examples].[Load_DimTractors_Flag_Records_For_Loading]**

6) A stored procedure that is called by the main routine and handles any deleting of dimension records - **[Examples].[Load_DimTractors_HandleDeletedRecords]**

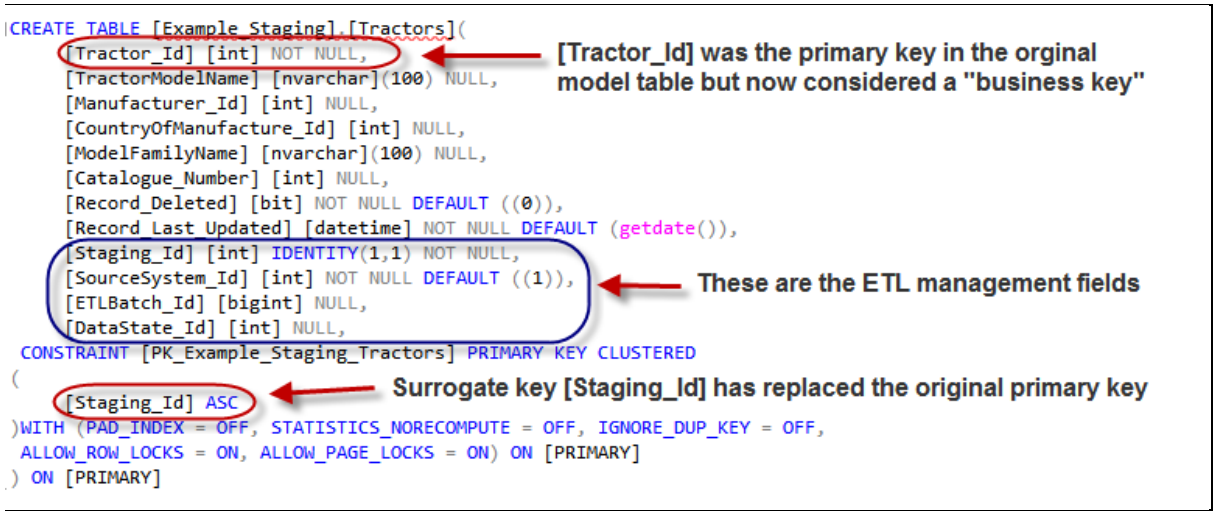


The generated staging table [Example_Staging].[Tractors]

This table combines the columns from the model table with some extra columns for ETL management.

You do not need to consider these ETL fields when you populate the staging table from your external source system with fresh data updates.

- **Staging_Id** is an IDENTITY column,
- **ETLBatch_Id** and **DataState_Id** are both nullable and managed by the generated ETL stored procedures and
- **SourceSystem_Id** has a usable default (1). You only need think about this if you have multiple systems feeding data into the one staging table and there is the need to avoid primary key conflicts.




```
CREATE TABLE [Example_Staging].[Tractors](
    [Tractor_Id] [int] NOT NULL,
    [TractorModelName] [nvarchar](100) NULL,
    [Manufacturer_Id] [int] NULL,
    [CountryOfManufacture_Id] [int] NULL,
    [ModelFamilyName] [nvarchar](100) NULL,
    [Catalogue_Number] [int] NULL,
    [Record_Deleted] [bit] NOT NULL DEFAULT ((0)),
    [Record_Last_Updated] [datetime] NOT NULL DEFAULT (getdate()),
    [Staging_Id] [int] IDENTITY(1,1) NOT NULL,
    [SourceSystem_Id] [int] NOT NULL DEFAULT ((1)),
    [ETLBatch_Id] [bigint] NULL,
    [DataState_Id] [int] NULL,
 CONSTRAINT [PK_Example_Staging_Tractors] PRIMARY KEY CLUSTERED
(
    [Staging_Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
 ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

The column **[SourceSystem_Id]** is to record the actual system each record originated in as you might now or in the future be extracting from many instances of the same kind of system.

For example, consider extracting data from 60 medical centres that all use the same practice management system. In this example, each system might be using the same range of values for their primary keys and when you amalgamate the records you could end up overwriting unrelated records due to key conflict.

Using the **[ETL].[SourceSystems]** table, you can create records to assign each medical centre a unique number. When you then load the data from each medical centre into the staging table, you also write in the respective SourceSystem_Id. The original primary key in your model table in combination with the SourceSystem_Id column forms the “business key” for your records. This will be enforced as a unique index on both the staging and dimension tables.

There is a placeholder value for SourceSystem_Id in the **[ETL].[SourceSystems]** table with a value of 1.

That is also the default value for the **[SourceSystem_Id]** in the generated staging table so if you only have one source system that feeds the staging table, you don’t need to think about it.

The column **[ETLBatch_Id]** records the batch (ETL cycle) that the record is flagged to be loaded within. The ETL driver routine for this staging table is [Examples].[Load_DimTractors] and it will load data in a series of batched cycles until the staging table is empty.

The column **[Datastate_Id]** indicates where each record is at in terms of the data load.

1) Staging - Flagged for load into dimension within batch
2) Staging - Loaded successfully into dimension within batch, ready for deletion
3) Staging - Error occurred loading records into dimension within batch
4) Dimension - Partial load of non-nullable columns (Part 1 of 2) - Not ready for reporting
5) Dimension - Load of all batched records complete (Part 2 of 2) - Ready for reporting
6) Dimension - Frozen, cannot be updated (Can only be set manually)

If the ETL fails to load records in a certain batch, they will be left in the staging table with a data state of 3.

Unlike records that were successfully loaded, these will not be deleted so you can investigate the problem. Failed records will also be skipped over when future ETL cycles run as these cycles only work on records for which **[ETLBatch_Id]** is initially NULL.

In example 9, we will force an error and show how to use the logs to determine what went wrong and how to fix it.

The generated dimension table [Examples].[DimTractors]

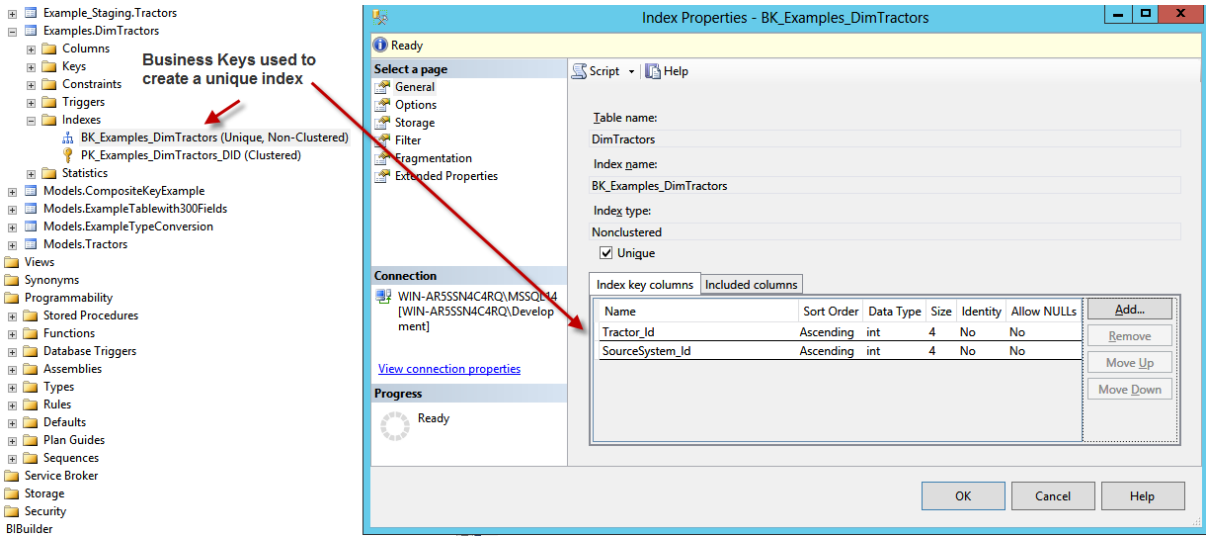
This is the dimension table built in the current example:

```
CREATE TABLE [Examples].[DimTractors](
     [DimTractors_did] [bigint] IDENTITY(1,1) NOT NULL,
     [Catalogue_Number] [int] NULL,
     [CountryOfManufacture_Id] [int] NULL,
     [Manufacturer_Id] [int] NULL,
     [Model_Family_Name] [nvarchar](512) NULL,
     [Record_Deleted] [bit] NOT NULL,
     [Record_Last_Updated] [datetime] NOT NULL,
     [Tractor_Id] [int] NOT NULL,
     [Tractor_Model_Name] [nvarchar](512) NULL,
     [SourceSystem_Id] [int] NOT NULL,
     [ETLBatch_Id] [int] NULL,
     [DataState_Id] [int] NULL,
 CONSTRAINT [PK_Examples_DimTractors_DID] PRIMARY KEY CLUSTERED
(
     [DimTractors_did] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF
    , IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
    ALLOW_PAGE_LOCKS = ON) ON [Examples]
) ON [Examples]

GO
```

Surrogate key with standardised name - Dim<table>_did

DID is a suffix used in BI Builder clearly indicating that column is a surrogate primary key for a dimension

Table created on the FILEGROUP specified in the example parameters



Column Metadata

If you would like to look at the metadata used in this example ELT build, run the SQL below:

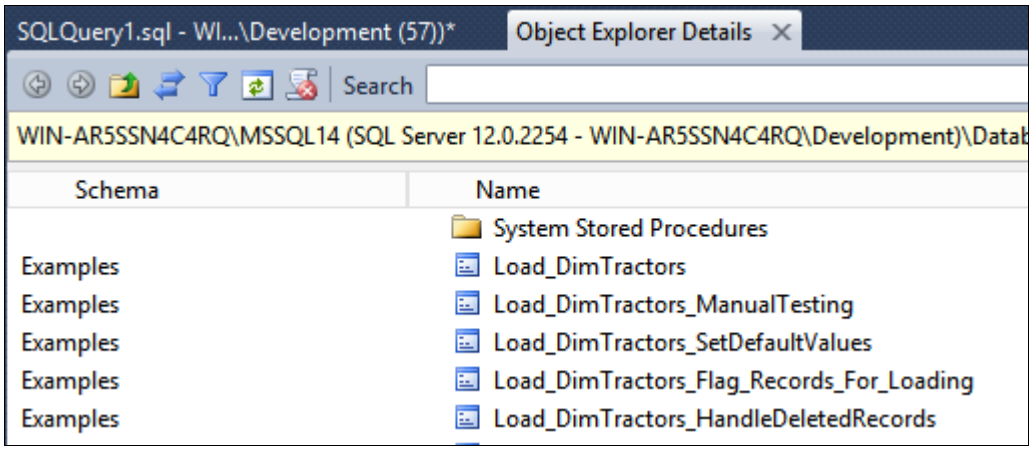
```
SELECT
      [StagingColumnName]
     ,[DataType]
     ,[CharacterMaximumLength]
     ,[DimensionColumnName]
     ,[DimensionDataType]
     ,[DimensionDataTypeLength]
     ,[OrdinalPosition]
     ,[ColumnDefault]
     ,[IncludeInDWH]
     ,[UseColumnForSCD]
     ,[IsBusinessKey]
     ,[IsDataCaptureDateTime]
     ,[IsDeletedFlag]
 FROM [BIB].[ColumnDefinitions]
 WHERE
       [StagingTableSchema]        = 'Example_Staging'
       AND
       [StagingTableName]    = 'Tractors'
 ORDER BY
       OrdinalPosition
```

The generated ETL Stored Procedures

BI Builder has generated four stored procedures to manage the ETL between staging and dimension tables.

There is only one of these that you need to execute to run the ETL – **Load_DimTractors**.

The fifth stored procedure named **Load_DimTractors_ManualTesting** is a routine you can run to perform various tests on your newly created dimension. In the header for the main ETL routine, **Load_DimTractors,** BI Builder has placed a script for testing the ETL using that fifth procedure.



Right click on **Load_DimTractors** in the SMSS Object Explorer window and select "*Script Stored Procedure as Create to*" to open it up in a query window for inspection.

Skip over the testing script in the header and scroll down to the start of the stored procedure definition.

```
CREATE PROCEDURE [Examples].[Load_DimTractors]
 AS
BEGIN
--*************************************************************************************
 -- This is an ETL routine that conducts data (new or updated or deleted as well as possibly sta
 -- from the staging table [Example_Staging].[Tractors] to the
 -- dimension table [Examples].[DimTractors]
 ------------------------------------------------------------------------------------------

 SET NOCOUNT ON
 SET DATEFORMAT dmy

-- Ensure that the dimension has an unknown "-1" key record holding default values
 -- for when the fact table has no proper key into the dimension
 DECLARE @ErrorCode int
 SET @ErrorCode = 0
 EXECUTE @ErrorCode = [Examples].[Load_DimTractors_SetDefaultValues]
IF @ErrorCode <> 0
BEGIN
     RETURN
 END
```

The stored procedure **[Examples].[Load_DimTractors_SetDefaultValues]** is called to ensure that there is a default record in place for when fact tables have rows that are NULL and do not link to any genuine row of the dimension table.

```
SELECT [DimTractors_did]
       ,[Catalogue_Number]
       ,[CountryOfManufacture_Id]
       ,[Manufacturer_Id]
       ,[Model_Family_Name]
       ,[Record_Deleted]
       ,[Record_Last_Updated]
       ,[Tractor_Id]
       ,[Tractor_Model_Name]
       ,[SourceSystem_Id]
       ,[ETLBatch_Id]
       ,[DataState_Id]
   FROM [Examples].[DimTractors]
 GO
```

This is the default dimension record created by the stored procedure [Load_DimTractors_SetDefaultValues]

Results | Messages

DimTractors_did	Catalogue_Number	CountryOfManufacture_Id	Manufacturer_Id	Model_Family_Name
-1	NULL	NULL	NULL	NULL
1	1055	1	1	Fieldmaster
2	2054	2	2	Mudblaster
3	766	1	1	Computracto

The next part of the stored procedure **Load_DimTractors** starts the looping in which a batch of records are selected from the staging table to be loaded. When there are no more records in staging with ETLBatch_Id = NULL, then @UnloadedCount = 0 and the loading process finishes.

```
DECLARE @UnloadedCount int
SET @UnloadedCount = (SELECT COUNT(*) FROM [Example_Staging].[Tractors] WHERE ETLBatch_Id IS NULL)
WHILE @UnloadedCount > 0
    BEGIN
```

The next clause initiates the logging for this ETL batch cycle and allocates the Batch ID.

```
SET @ProcessName = '[Examples].[Load_DimTractors]'
EXECUTE @Batch_ID = [ETL].[InitiateETLBatch] @ProcessName
```

Using the Batch ID, a certain number of records in staging are flagged for loading. By default, the batch size is 10,000 records to load in each cycle. You can configure this in the **[BIB_Settings]** table and you can also override this in the generated procedure **Load_DimTractors_Flag_Records_For_Loading**.

```
EXECUTE @StepId = [etl].[AddETLBatchStepStatus] @Batch_ID , 2 , 'Flagging Started ...' , NULL;
 ;WITH CandidateRecords
    AS (
    SELECT TOP 10000 Staging_ID
           FROM [Example_Staging].[Tractors]
           WHERE
            [ETLBatch_Id] IS NULL
           ORDER BY Staging_ID ASC
    )
    UPDATE s
    SET [ETLBatch_Id]    = @Batch_ID ,
        [DataState_Id]   = 1
        FROM Example_Staging.Tractors s
             INNER JOIN   CandidateRecords c ON c.Staging_Id = s.Staging_Id

    SET @RecordsAffected = @@ROWCOUNT
```

Override this value in the generated stored procedure Load_DimTractors_Flag_Records_For_Loading if you want to tune it for better performance or a lighter footprint

The next part of the stored procedure **Load_DimTractors** handles any records in the current batch that have been soft deleted in the source system**.**

```
EXECUTE [Examples].[Load_DimTractors_HandleDeletedRecords] @Batch_ID
```

BI Builder attempts to detect any columns that appear to be deletion flags. It does this in the `[BIB].[FlagDeletedBitFieldInStagingTable]` stored procedure.

If your data sources conform to a particular naming convention around columns used for soft deletes, you can ensure that this routine below detects them and flags them in the metadata used for the ETL build.

This detection routine currently depends on the "deleted" field being a BIT field. This field is encoded into the ETL stored procedures to both delete existing expired records from the dimension (if desired) and also to ensure deleted records are not entered into the dimension.

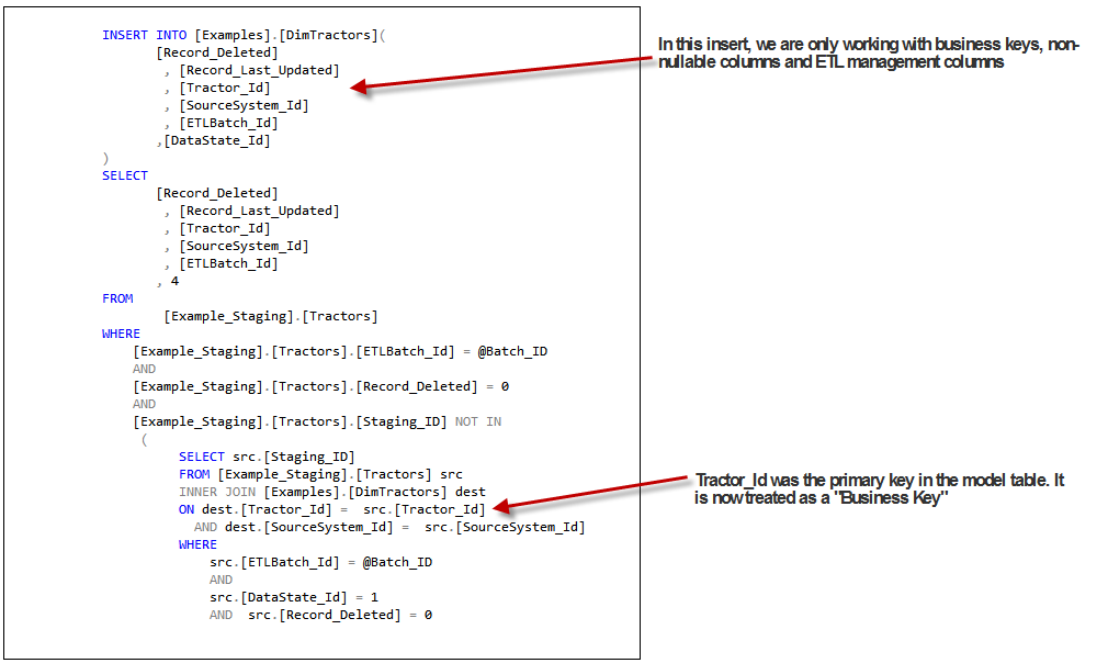
If there is no column to indicate if a record has been deleted in the source system, the generated ETL will still function.

Back in the main stored procedure **Load_DimTractors**, it performs an insert of any records that are new to the dimension.

The pattern used in refreshing the dimension is to

A) Insert non-nullable fields that are new (based on the business keys) and then
B) Update all non-key records.

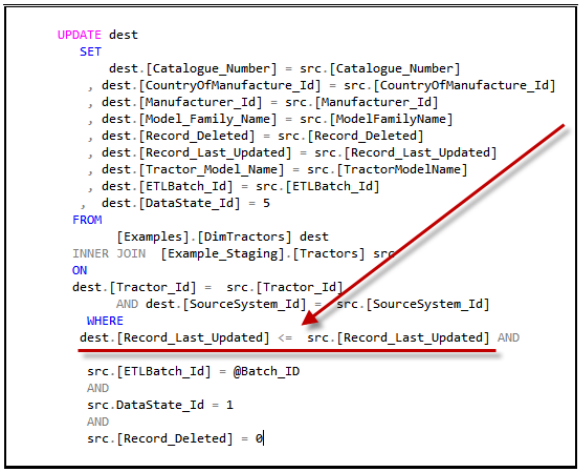
You might have other approaches to refreshing dimensions – perhaps using the MERGE statement. You can edit the various routines that build the ETL to reflect your preferred approach if BI Builder's default approach does not suit you.




```
INSERT INTO [Examples].[DimTractors](
       [Record_Deleted]
        , [Record_Last_Updated]
        , [Tractor_Id]
        , [SourceSystem_Id]
        , [ETLBatch_Id]
       ,[DataState_Id]
)
SELECT
       [Record_Deleted]
        , [Record_Last_Updated]
        , [Tractor_Id]
        , [SourceSystem_Id]
        , [ETLBatch_Id]
       , 4
FROM
        [Example_Staging].[Tractors]
WHERE
    [Example_Staging].[Tractors].[ETLBatch_Id] = @Batch_ID
    AND
    [Example_Staging].[Tractors].[Record_Deleted] = 0
    AND
    [Example_Staging].[Tractors].[Staging_ID] NOT IN
     (
          SELECT src.[Staging_ID]
          FROM [Example_Staging].[Tractors] src
          INNER JOIN [Examples].[DimTractors] dest
          ON dest.[Tractor_Id] =  src.[Tractor_Id]
            AND dest.[SourceSystem_Id] =  src.[SourceSystem_Id]
          WHERE
              src.[ETLBatch_Id] = @Batch_ID
              AND
              src.[DataState_Id] = 1
              AND  src.[Record_Deleted] = 0
```

[DataState_Id] is set to "(4) *Dimension - Partial load of non-nullable columns (Part 1 of 2) - Not ready for reporting*" and if the load fails before *all* columns are updated, these (4) records will be removed from the dimension as they are not complete records.

To complete the data load, an UPDATE statement is executed next – this updates all the non-key fields in records that were either inserted in the previous clause or where an older version of the record already exists in the dimension.




```
UPDATE dest
   SET
       dest.[Catalogue_Number] = src.[Catalogue_Number]
     , dest.[CountryOfManufacture_Id] = src.[CountryOfManufacture_Id]
     , dest.[Manufacturer_Id] = src.[Manufacturer_Id]
     , dest.[Model_Family_Name] = src.[ModelFamilyName]
     , dest.[Record_Deleted] = src.[Record_Deleted]
     , dest.[Record_Last_Updated] = src.[Record_Last_Updated]
     , dest.[Tractor_Model_Name] = src.[TractorModelName]
     , dest.[ETLBatch_Id] = src.[ETLBatch_Id]
     ,  dest.[DataState_Id] = 5
  FROM
        [Examples].[DimTractors] dest
  INNER JOIN  [Example_Staging].[Tractors] src
  ON
  dest.[Tractor_Id] =  src.[Tractor_Id]
        AND dest.[SourceSystem_Id] =  src.[SourceSystem_Id]
    WHERE
   dest.[Record_Last_Updated] <=  src.[Record_Last_Updated] AND

    src.[ETLBatch_Id] = @Batch_ID
    AND
    src.DataState_Id = 1
    AND
    src.[Record_Deleted] = 0
```

The final step within a batch cycle is to delete records from staging that were successfully processed in the ETL.

The procedure loops through this INSERT – UPDATE cycle on batches of staged records until the staging table is empty of records.

Testing the new ETL procedures

Scroll to the top of the [Load_DimTractors] stored procedure and you will see a prepared script you can use to test the newly created ETL.

```
/*
--=============================================================================================================
-- MANUAL TESTING UTILITY ROUTINES FOR [Examples].[Load_DimTractors]
-------------------------------------------------------------------------
-- Records in Dimension
SELECT * FROM      [Examples].[DimTractors]
SELECT COUNT(*) FROM [Examples].[DimTractors]

-- Records in Staging
SELECT * FROM       [Example_Staging].[Tractors]
SELECT COUNT(*) FROM  [Example_Staging].[Tractors]

-- To start from absolute scratch with no pre-existing logs, restaged records in the staging table and an em
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,1,1,1,1

-- See the logs of previous ETL runs
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,0,1

-- Clear the logs for this dimension
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,0,0,0,0

-- Clear the logs, restage the test data and truncate the dimension
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,1,1,0,0

-- To restage data in the staging table
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,1,0,0

-- To run process on whatever is in the staging table and show log
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,1,1

-- To Rerun the process on restaged data in the staging table
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,1,1,1

-- To Rerun the process on previously ingested data in the staging table
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,1,1,1,0 -- reset and run first, no log report
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,1,1,1 -- refill staging and run load routine again,
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,1,1 -- try reingesting staged data again - stagir
```

Your starting position is an empty dimension and a staging table holding the data you have placed in your model table.

You could begin then by simply executing [Examples].[Load_DimTractors] as you would in a routine that drives your complete ETL – however, we will use the manual testing procedure in order to immediately see the logs of the operation.

Start by confirming your staging table has data by executing either of these queries:

```
-- Records in Staging
SELECT * FROM      [Example_Staging].[Tractors]
SELECT COUNT(*) FROM  [Example_Staging].[Tractors]
```

Now run the data load and view the log by executing this statement:

```
-- To run process on whatever is in the staging table and show log
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,1,1
```

Results | Messages

	ETL_Batch_Log_Id	ETL_Batch_Details_Id	Process_Name	Start_Time	ETLLoadStatus_Id	End_Time	Run_Duration	Duration	Success_Flag	Error_Code	RecordsAffected	Message
1	4	20	[Examples].[Load_DimTractors]	2014-10-13 15:58:35.860	4	2014-10-13 15:58:35.893	0:00:00	0.00	1	NULL	3	Updating records succeeded
2	4	19	[Examples].[Load_DimTractors]	2014-10-13 15:58:35.860	3	2014-10-13 15:58:35.883	0:00:00	0.02	1	NULL	3	Inserting new records succeeded
3	4	18	[Examples].[Load_DimTractors]	2014-10-13 15:58:35.860	4	2014-10-13 15:58:35.877	0:00:00	0.02	1	NULL	0	Deleting flagged records succeeded
4	4	17	[Examples].[Load_DimTractors]	2014-10-13 15:58:35.860	2	2014-10-13 15:58:35.867	0:00:00	0.02	1	NULL	3	Flagging operation succeeded
5	4	16	[Examples].[Load_DimTractors]	2014-10-13 15:58:35.860	1	2014-10-13 15:58:35.863	0:00:00	0.00	1	NULL	0	ETL Batch Initiated OK

The log shows us that:

A) There were no errors encountered
B) There were 3 records flagged in a single batch for loading
C) Of those records, none were “deletions”
D) As the dimension was empty, all records are new and there were 3 inserts
E) There were 3 updates.

The staging table should now be empty. You can confirm that using:

```
-- Records in Staging
SELECT * FROM      [Example_Staging].[Tractors]
SELECT COUNT(*) FROM  [Example_Staging].[Tractors]
```

The dimension table should hold those 3 records as well as the default record (DID key = -1). You can confirm that using:

```
-- Records in Dimension
SELECT * FROM     [Examples].[DimTractors]
SELECT COUNT(*) FROM [Examples].[DimTractors]
```

Results | Messages

	DimTractors_did	Catalogue_Number	CountryOfManufacture_Id	Manufacturer_Id	Model_Family_Name
1	-1	NULL	NULL	NULL	NULL
2	1	1055	1	1	Fieldmaster
3	2	2054	2	2	Mudblaster
4	3	766	1	1	Computracto

We can now refill staging with the same records it held previously using the following SQL statement:

```
-- To restage data in the staging table
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,1,0,0
```

If we now re-run the ETL, we would expect to have no inserts – as the same records were loaded just before – and we expect 3 updates as the records have the same modified dates as the previous load.

```
-- To run process on whatever is in the staging table and show log
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,1,1
```

The log will show the new ETL cycle but this time the records affected for the INSERT action will be 0 and the UPDATES are 3.

Staging is now empty – run the above routine again and you will see it takes no action and no new logs are created. Logs are only created when there is data available in staging.

Now we will test deletions.

```
-- To test deletions

EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,1,1,1,0 -- reset and run first, no log report

EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,1,0,0 -- restage the data


UPDATE   [Example_Staging].[Tractors]
    SET [Record_Deleted] = 1 WHERE Staging_Id = 1 -- flag the first record as deleted in staging


EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,1,0 -- run the load again

EXECUTE   [Examples].[Load_DimTractors_ManualTesting] 0,0,0,0,1 -- check the logs to see that
                                        -- records were inserted and updated, then in one
                                        -- batch there was one deletions and 2 updates

SELECT * FROM       [Examples].[DimTractors] -- View the records in the dimension to ensure the
                                        -- flagged record(s) have been deleted
```

Results | Messages

	ETL_Batch_Log_Id	ETL_Batch_Details_Id	Success_Flag	Error_Code	RecordsAffected	Message
1	7	35	1	NULL	2	Updating records succeeded
2	7	34	1	NULL	0	Inserting new records succeeded
3	7	33	1	NULL	1	Deleting flagged records succeeded
4	7	32	1	NULL	3	Flagging operation succeeded
5	7	31	1	NULL	0	ETL Batch Initiated OK
6	6	30	1	NULL	3	Updating records succeeded
7	6	29	1	NULL	3	Inserting new records succeeded
8	6	28	1	NULL	0	Deleting flagged records succeeded
9	6	27	1	NULL	3	Flagging operation succeeded
10	6	26	1	NULL	0	ETL Batch Initiated OK

2nd Run

1st Run

	DimTractors_did	Catalogue_Number	CountryOfManufacture_Id	Manufacturer_Id	Model_Family_Name	Record_Deleted	Record_Las
1	-1	NULL	NULL	NULL	NULL	0	1900-01-01
2	2	2054	2	2	Mudblaster	0	2014-09-23
3	3	766	1	1	Computracto	0	2014-09-23

That was a detailed view of what BI Builder does using a simple example.

The following examples are brief and show single facets of BI Builder usage.

Example 2: A Type 2 Slowly Changing Dimension

In this example, we will create a Type 2 SCD in which changes to two columns will trigger creation of a new active record in the dimension and the old records are preserved.

The build script for this example is almost identical to the first example as it feeds off the same model table (Tractors) except

A) We have specified the SCD "trigger" columns as

```
SET @SlowlyChangingDimensionColumnList =
'Manufacturer_Id,CountryOfManufacture_Id'
```

There are no spaces in this comma-delimited list and the column names are not square-bracketed.

B) We have changed the name of the staging and dimension tables to preserve the ones we made in Example 1.

The complete ETL Build Script is:

```
-- Example 2 A Type 2 SCD Dimension
-- ETL BUILD SCRIPT FOR TABLE [Models].[Tractors]


DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @SourceFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'Tractors'

-- Staging Table Details ------------
    SET @SourceFilegroupName = 'Example_Staging'
    SET @StagingSchemaName = 'Example_Staging'
    SET @StagingTableName = 'TractorsSCDType2'
    SET @SlowlyChangingDimensionColumnList = 'Manufacturer_Id,CountryOfManufacture_Id'

-- Dimension Table Details ------------
    SET @DimSchemaName = 'Examples'
    SET @DimTableName = 'DimTractorsSCDType2'
    SET @DimFilegroupName  = 'Examples'
    SET @SourceSystem_Id  = 1

-- Control Flags for BI Builder Main()
    SET @SkipDataAnalysis = 0
    SET @ApplyAutoAudit = 0

    ---- 1) Control Flags for automated end-to-end build process
    SET @PauseForManualEditsToMetadata = 0
    SET @UseCurrentMetadataForBuild = 0

EXECUTE[bib].[Main]
   @ModelSchemaName
  ,@ModelTableName
  ,@StagingSchemaName
  ,@StagingTableName
  ,@SourceFilegroupName
  ,@SourceSystem_Id
  ,@SlowlyChangingDimensionColumnList
  ,@DimSchemaName
  ,@DimTableName
  ,@DimFilegroupName
  ,@SkipDataAnalysis
  ,@PauseForManualEditsToMetadata
  ,@UseCurrentMetadataForBuild
  ,@ApplyAutoAudit

GO
```

In the table **[BIB].[ColumnDefinitions]**, those columns are flagged as SCD and when the dimension is built it has the fields required to support SCD functionality.

```
CREATE TABLE [Examples].[DimTractorsSCDType2](
	[DimTractorsSCDType2_did] [bigint] IDENTITY(1,1) NOT NULL,
	[Catalogue_Number] [int] NULL,
	[CountryOfManufacture_Id] [int] NULL,
	[Manufacturer_Id] [int] NULL,
	[Model_Family_Name] [nvarchar](512) NULL,
	[Record_Deleted] [bit] NOT NULL,
	[Record_Last_Updated] [datetime] NOT NULL,
	[Tractor_Id] [int] NOT NULL,
	[Tractor_Model_Name] [nvarchar](512) NULL,
	[SourceSystem_Id] [int] NOT NULL,
	[ETLBatch_Id] [int] NULL,
	[DataState_Id] [int] NULL,
	[EffectiveStart] [datetime] NOT NULL,
	[EffectiveFinish] [datetime] NULL,
	[ActiveRecord] [bit] NOT NULL,
 CONSTRAINT [PK_Examples_DimTractorsSCDType2_DID] PRIMARY KEY CLUSTERED
(
	[DimTractorsSCDType2_did] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [Examples]
) ON [Examples]

GO

ALTER TABLE [Examples].[DimTractorsSCDType2] ADD  DEFAULT ((1)) FOR
[SourceSystem_Id]
GO

ALTER TABLE [Examples].[DimTractorsSCDType2] ADD  DEFAULT (getdate()) FOR
[EffectiveStart]
GO

ALTER TABLE [Examples].[DimTractorsSCDType2] ADD  DEFAULT ((1)) FOR [ActiveRecord]
GO
```

If you look into the stored procedure **[Examples].[Load_DimTractorsSCDType2]**, you will see various sections of logic relating to the SCD columns – below is an illustration from the INSERT statement that runs within each batch cycle. It is effectively using the SCD columns you nominated as it does business keys.

```
FROM
        [Example_Staging].[TractorsSCDType2]

WHERE
    [Example_Staging].[TractorsSCDType2].[ETLBatch_Id] = @Batch_ID
    AND
    [Example_Staging].[TractorsSCDType2].[Record_Deleted] = 0
    AND
    [Example_Staging].[TractorsSCDType2].[Staging_ID] NOT IN
     (
          SELECT src.[Staging_ID]
          FROM [Example_Staging].[TractorsSCDType2] src
          INNER JOIN [Examples].[DimTractorsSCDType2] dest
          ON dest.[CountryOfManufacture_Id] =  src.[CountryOfManufacture_Id]
            AND dest.[Manufacturer_Id] =  src.[Manufacturer_Id]
            AND dest.[Tractor_Id] =  src.[Tractor_Id]
            AND dest.[SourceSystem_Id] =  src.[SourceSystem_Id]
          WHERE
              src.[ETLBatch_Id] = @Batch_ID
              AND
              src.[DataState_Id] = 1
              AND  src.[Record_Deleted] = 0
```

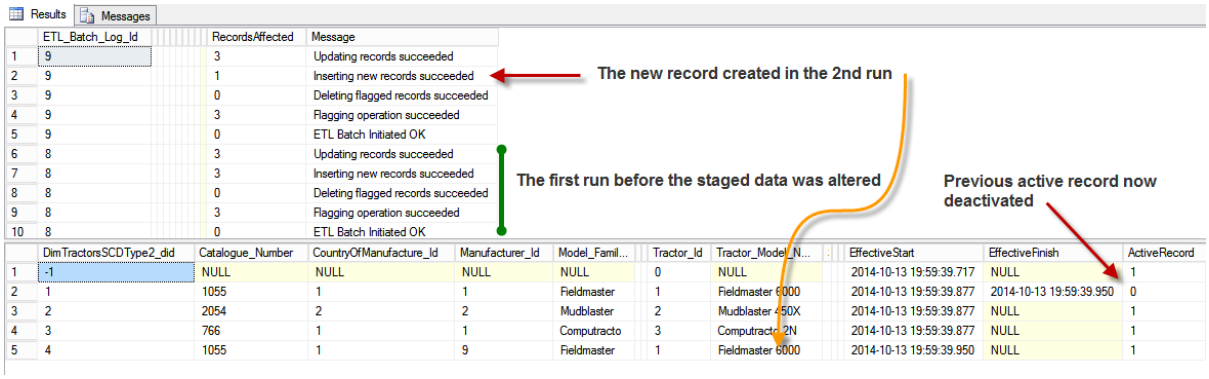
Example from the INSERT clause showing how the SCD columns are employed to determine if a new record should be inserted

You will notice in the header of the procedure **[Examples].[Load_DimTractorsSCDType2]** that there is now amongst the test routines a script to give the SCD function a go.

```
-- To test SCD functionality

EXECUTE   [Examples].[Load_DimTractorsSCDType2_ManualTesting] 1,1,1,1,0 -- reset and
run first, no log report

EXECUTE   [Examples].[Load_DimTractorsSCDType2_ManualTesting] 0,0,1,0,0 -- restage
the data

-- Adjust one of the SCD columns
UPDATE   [Example_Staging].[TractorsSCDType2]
    SET [Manufacturer_Id] = 9 -- to trigger a new record created in the dimension
when loaded
          , [Record_Last_Updated] = GETDATE()
    FROM    [Example_Staging].[TractorsSCDType2]
    WHERE Staging_Id = 1

EXECUTE   [Examples].[Load_DimTractorsSCDType2_ManualTesting] 0,0,0,1,0 -- run the
load

EXECUTE    [Examples].[Load_DimTractorsSCDType2_ManualTesting] 0,0,0,0,1 -- check
the logs to see that records were inserted and updated, should be 1 insert and 1
update

SELECT * FROM     [Examples].[DimTractorsSCDType2] -- check out the records to
ensure the EffectiveFinish date and Active record flags have been set correctly
```

The value 9 is one the author plugged in to do the testing – it is simply a value different to what the record held previously in that column.

Run the script and you will see you now have a 5th record in the dimension and it has become the active one for its primary key. The previous record with that primary key is marked as inactive and has its **[EffectiveFinish]** date filled in.



	ETL_Batch_Log_Id	RecordsAffected	Message
1	9	3	Updating records succeeded
2	9	1	Inserting new records succeeded
3	9	0	Deleting flagged records succeeded
4	9	3	Flagging operation succeeded
5	9	0	ETL Batch Initiated OK
6	8	3	Updating records succeeded
7	8	3	Inserting new records succeeded
8	8	0	Deleting flagged records succeeded
9	8	3	Flagging operation succeeded
10	8	0	ETL Batch Initiated OK

	DimTractorsSCDType2_did	Catalogue_Number	CountryOfManufacture_Id	Manufacturer_Id	Model_Famil...	Tractor_Id	Tractor_Model_N...	EffectiveStart	EffectiveFinish	ActiveRecord
1	-1	NULL	NULL	NULL	NULL	0	NULL	2014-10-13 19:59:39.717	NULL	1
2	1	1055	1	1	Fieldmaster	1	Fieldmaster 6000	2014-10-13 19:59:39.877	2014-10-13 19:59:39.950	0
3	2	2054	2	2	Mudblaster	2	Mudblaster 450X	2014-10-13 19:59:39.877	NULL	1
4	3	766	1	1	Computracto	3	Computracto 2N	2014-10-13 19:59:39.877	NULL	1
5	4	1055	1	9	Fieldmaster	1	Fieldmaster 6000	2014-10-13 19:59:39.950	NULL	1

Example 3: Composite Keys

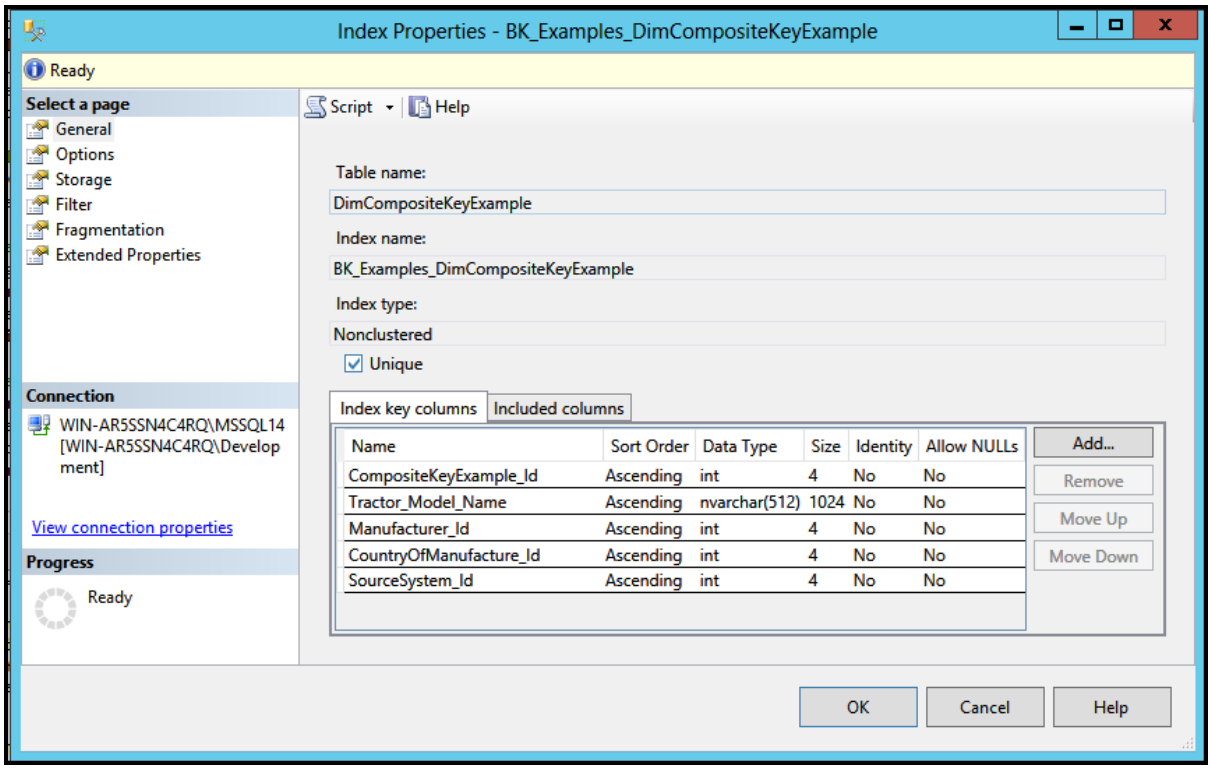
The example here is [Models].[CompositeKeyExample] and we can produce an ETL build script for it by executing the following SQL:

```
EXECUTE [BIB].[Main_PrintScriptToBuildETLFromModelTable] 'Models', 'CompositeKeyExample'
```

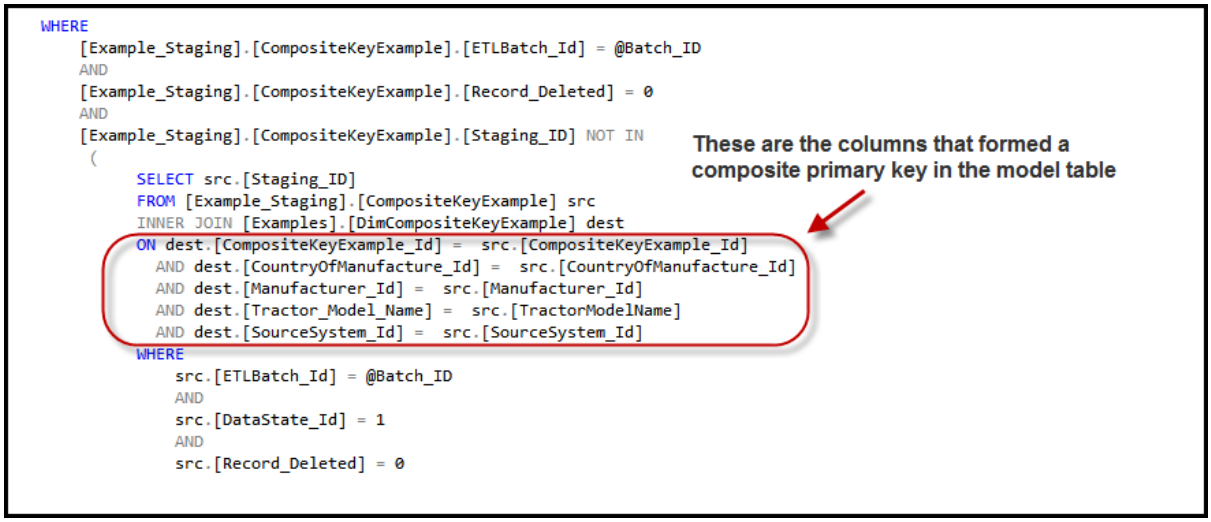
This produces the script below (except the author has filled in the placeholders for the names of the generated objects) – Execute this script to run the example build:

```
-- Example 3 - Composite Primary Keys on the model table
-- ETL BUILD SCRIPT FOR TABLE [Models].[CompositeKeyExample]
-----------------------------------------------------------------------------------
DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @StagingFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'CompositeKeyExample'

-- Staging Table Details ------------
    SET @StagingFilegroupName = 'Example_Staging'
    SET @StagingSchemaName = 'Example_Staging'
    SET @StagingTableName = 'CompositeKeyExample'
    SET @SlowlyChangingDimensionColumnList = NULL

-- Dimension Table Details ------------
    SET @DimSchemaName = 'Examples'
    SET @DimTableName = 'DimCompositeKeyExample'
    SET @DimFilegroupName  = 'Examples'
    SET @SourceSystem_Id  = 1

-- Control Flags for BI Builder Main()
    SET @SkipDataAnalysis = 0
    SET @ApplyAutoAudit = 0

    ---- 1) Control Flags for automated end-to-end build process
    SET @PauseForManualEditsToMetadata = 0
    SET @UseCurrentMetadataForBuild = 0

EXECUTE[bib].[Main]
   @ModelSchemaName
  ,@ModelTableName
  ,@StagingSchemaName
  ,@StagingTableName
  ,@StagingFilegroupName
  ,@SourceSystem_Id
  ,@SlowlyChangingDimensionColumnList
  ,@DimSchemaName
  ,@DimTableName
  ,@DimFilegroupName
  ,@SkipDataAnalysis
  ,@PauseForManualEditsToMetadata
  ,@UseCurrentMetadataForBuild
  ,@ApplyAutoAudit

GO
```

If you inspect the generated dimension table [Examples].[DimCompositeKeyExample], you will see the four primary keys from the original table are now forming a unique index:



Now view the freshly generated stored procedure **[Examples].[Load_DimCompositeKeyExample]** and you will see in various places the role these business keys play in data loading:




```
WHERE
    [Example_Staging].[CompositeKeyExample].[ETLBatch_Id] = @Batch_ID
    AND
    [Example_Staging].[CompositeKeyExample].[Record_Deleted] = 0
    AND
    [Example_Staging].[CompositeKeyExample].[Staging_ID] NOT IN
     (
          SELECT src.[Staging_ID]
          FROM [Example_Staging].[CompositeKeyExample] src
          INNER JOIN [Examples].[DimCompositeKeyExample] dest
          ON dest.[CompositeKeyExample_Id] =  src.[CompositeKeyExample_Id]
            AND dest.[CountryOfManufacture_Id] =  src.[CountryOfManufacture_Id]
            AND dest.[Manufacturer_Id] =  src.[Manufacturer_Id]
            AND dest.[Tractor_Model_Name] =  src.[TractorModelName]
            AND dest.[SourceSystem_Id] =  src.[SourceSystem_Id]
          WHERE
              src.[ETLBatch_Id] = @Batch_ID
              AND
              src.[DataState_Id] = 1
              AND
              src.[Record_Deleted] = 0
```

Scroll to the top of the stored procedure **[Examples].[Load_DimCompositeKeyExample]** and you see the testing routines – run the following to confirm that the generated ETL works as expected:

```
-- To run process on whatever is in the staging table and show log
EXECUTE  [Examples].[Load_DimCompositeKeyExample_ManualTesting] 0,0,0,1,1
```

Example 4: Custom Type Conversions

In this example, we have a date column in the example source table **[Models].[ExampleTypeConversion]** that is expressed as the number of seconds since 1970. We want our generated dimension table to instead express this value as a DateTime.

We start by generating a build script using the developer utility routine:

```
EXECUTE [BIB].[Main_PrintScriptToBuildETLFromModelTable] 'Models', 'ExampleTypeConversion'
```

Fill in the names of your generated objects to end up with a script like this:

```
DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @StagingFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'ExampleTypeConversion'
-- Staging Table Details ------------
    SET @StagingFilegroupName = 'Example_Staging'
    SET @StagingSchemaName = 'Example_Staging'
    SET @StagingTableName = 'ExampleTypeConversion'
    SET @SlowlyChangingDimensionColumnList = NULL
-- Dimension Table Details ------------
    SET @DimSchemaName = 'Examples'
    SET @DimTableName = 'DimExampleTypeConversion'
    SET @DimFilegroupName  = 'Examples'
    SET @SourceSystem_Id  = 1
-- Control Flags for BI Builder Main()
    SET @SkipDataAnalysis = 0
    SET @ApplyAutoAudit = 0
    ---- 1) Control Flags for automated end-to-end build process
    SET @PauseForManualEditsToMetadata = 0
    SET @UseCurrentMetadataForBuild = 0

EXECUTE[bib].[Main]
   @ModelSchemaName
  ,@ModelTableName
  ,@StagingSchemaName
  ,@StagingTableName
  ,@StagingFilegroupName
  ,@SourceSystem_Id
  ,@SlowlyChangingDimensionColumnList
  ,@DimSchemaName
  ,@DimTableName
  ,@DimFilegroupName
  ,@SkipDataAnalysis
  ,@PauseForManualEditsToMetadata
  ,@UseCurrentMetadataForBuild
  ,@ApplyAutoAudit

GO
```

Execute this script and you will see that the staging column [Integer_Style_Timestamp] in the staging table has been converted to a DateTime field in the dimension table and the ETL stored procedure performs the data type transformation using a scalar function.

```
CREATE TABLE [Examples].[DimExampleTypeConversion](
    [DimExampleTypeConversion_did] [bigint] IDENTITY(1,1) NOT NULL,
    [A_Text_Field] [nvarchar](512) NOT NULL,
    [ExampleTypeConversion_Id] [int] NOT NULL,
    [Integer_Style_Timestamp] [datetime] NOT NULL,
    [SourceSystem_Id] [int] NOT NULL,
    [ETLBatch_Id] [int] NULL,
    [DataState_Id] [int] NULL,
```

The dimension column is now a DateTime

In the generated stored procedure, the transformation is preformed where required:

```
    INSERT INTO [Examples].[DimExampleTypeConversion](
           [A_Text_Field]
            , [ExampleTypeConversion_Id]
            , [Integer_Style_Timestamp]
            , [SourceSystem_Id]
            , [ETLBatch_Id]
           ,[DataState_Id]
    )

    SELECT
              [ATextField]
            , [ExampleTypeConversion_Id]
            , [DWHUtils].[IntegerTimestampToDatetime]([Integer_Style_Timestamp])
            , [SourceSystem_Id]
            , [ETLBatch_Id]
           , 4
    FROM
            [Example_Staging].[ExampleTypeConversion]

    WHERE
```

Transformation from [int] to [DateTime] takes place in the stored procedure [Examples].[Load_DimExampleTypeConversion]

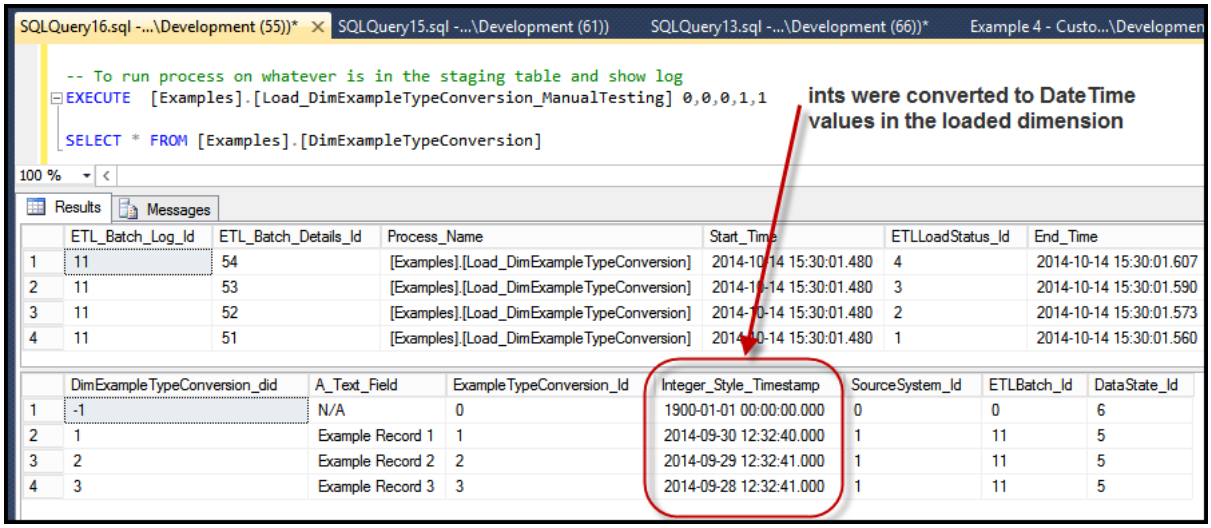
As this column was also detected as the Modified Date style column, the transformation is also supplied to the WHERE clause that protects fresh data from being overwritten with stale data.

```
  UPDATE dest
     SET
         dest.[A_Text_Field] = src.[ATextField]
       , dest.[Integer_Style_Timestamp] = [DWHUtils].[IntegerTimestampToDatetime](src.[Integer_Style_Timestamp])
       , dest.[ETLBatch_Id] = src.[ETLBatch_Id]
         , [DataState_Id] = 5
    FROM
          [Examples].[DimExampleTypeConversion] dest
    INNER JOIN  [Example_Staging].[ExampleTypeConversion] src
    ON
    dest.[ExampleTypeConversion_Id] =  src.[ExampleTypeConversion_Id]
          AND dest.[SourceSystem_Id] =  src.[SourceSystem_Id]
      WHERE
     dest.[Integer_Style_Timestamp] <=  [DWHUtils].[IntegerTimestampToDatetime](src.[Integer_Style_Timestamp]) AND

      src.[ETLBatch_Id] = @Batch_ID
      AND
      src.DataState_Id = 1
```

Conversions from int to DateTime

Now execute the ETL to test the conversion:

```
-- To run process on whatever is in the staging table and show log
EXECUTE  [Examples].[Load_DimExampleTypeConversion_ManualTesting] 0,0,0,1,1

SELECT * FROM [Examples].[DimExampleTypeConversion]
```



How to get your own type transformations into the ETL Build Process

There are two ways to achieve this – an automated way that applies to every build or an ad-hoc way in which you pause the build to do some tweaks.

You will see the automated way in Appendix 1 where details of the stored procedure **`[BIB].[SetCustomTypeConversionExpressionThroughRules]`** is discussed.

You would apply this approach if you have a data source where there are columns you want to transform that are spread throughout your data source tables and that have a small set of names - such that you can encode simple rules to detect them by column name and data type.

The ad-hoc way is shown in Example 11 – Pausing ETL Builds to edit Metadata.

Example 5: Column Name Transformations

BI Builder calls a scalar function during the ETL Build that can apply column name transformations.

This function is **`[BIB].[MakeDimensionColumnNameFromStagingColumn]`**

The function currently applies a few example transformation rules that you can replace with your own.

> **Do note that BI Builder does not alter any column names that end in "Id"** – as these are likely to be keys.
>
> Those columns are unlikely to be displayed to business report users but you as data developer will need to think about them often. Having BI Builder alter them will only increase the cognitive load on you. Also, the user interface in SQL editors such as SMSS will often join like-named columns when creating views and this can be quite handy.

If Salesforce is a data source you are familiar with, you will know of the naming convention whereby custom columns are suffixed with "__c". One example rule in the BIB function removes this suffix from the columns names that are created in the dimension table.

You might also have abbreviations used in your data source that could confuse the business users of reports. For example, 'Cust_St_Dt' might be known as meaning "Customer Start Date" but should read as "Customer Start Date" in reports to keep the user training required to a minimum.

You could always adjust the column name from within SSAS, SSRS or in queries that feed reports yet that could be time consuming. It could be better than the dimension column is named "Customer_Start_Date" and when loaded into SSAS, it will display as "Customer Start Date" automatically.

In this scalar function `[BIB].[MakeDimensionColumnNameFromStagingColumn]` you can apply your own text transformations to expand out abbreviations.

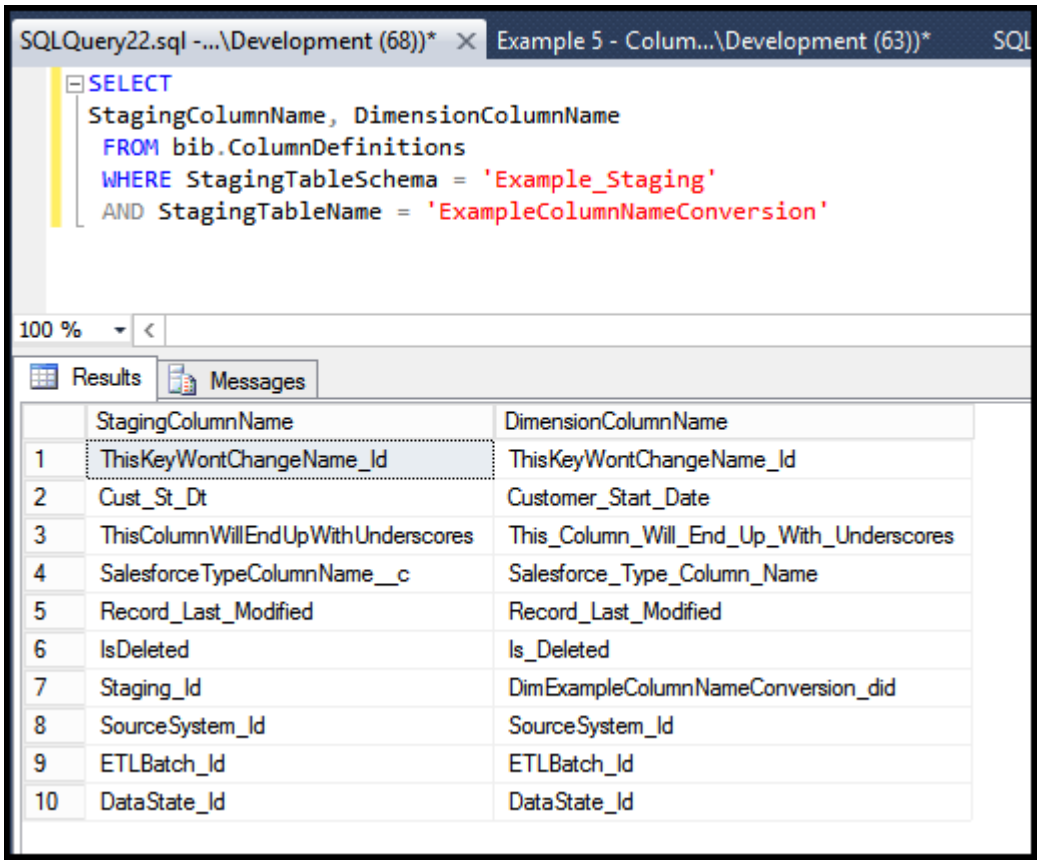
The routine also transforms PascalCased text to insert "_" between capital letters that follow lower case letters. So "CustomerSuburb" will be transformed in the dimension column name to "Customer_Suburb".

If you run the following script, you will see the example column name transformations clearly:

```
-- ETL BUILD SCRIPT FOR TABLE [Models].[ExampleColumnNameConversion]
DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @StagingFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'ExampleColumnNameConversion'

-- Staging Table Details ------------
    SET @StagingFilegroupName = 'Example_Staging'
    SET @StagingSchemaName = 'Example_Staging'
    SET @StagingTableName = 'ExampleColumnNameConversion'
    SET @SlowlyChangingDimensionColumnList = NULL

-- Dimension Table Details ------------
    SET @DimSchemaName = 'Examples'
    SET @DimTableName = 'DimExampleColumnNameConversion'
    SET @DimFilegroupName  = 'Examples'
    SET @SourceSystem_Id  = 1

-- Control Flags for BI Builder Main()
    SET @SkipDataAnalysis = 0
    SET @ApplyAutoAudit = 0

    ---- 1) Control Flags for automated end-to-end build process
    SET @PauseForManualEditsToMetadata = 0
    SET @UseCurrentMetadataForBuild = 0

EXECUTE[bib].[Main]
   @ModelSchemaName
  ,@ModelTableName
  ,@StagingSchemaName
  ,@StagingTableName
  ,@StagingFilegroupName
  ,@SourceSystem_Id
  ,@SlowlyChangingDimensionColumnList
  ,@DimSchemaName
  ,@DimTableName
  ,@DimFilegroupName
  ,@SkipDataAnalysis
  ,@PauseForManualEditsToMetadata
  ,@UseCurrentMetadataForBuild
  ,@ApplyAutoAudit

GO
```

Then run this query to view the metadata in [BIB].[ColumnDefinitions].

```
SELECT
StagingColumnName, DimensionColumnName
 FROM bib.ColumnDefinitions
 WHERE StagingTableSchema = 'Example_Staging'
 AND StagingTableName = 'ExampleColumnNameConversion'
```



	StagingColumnName	DimensionColumnName
1	ThisKeyWontChangeName_Id	ThisKeyWontChangeName_Id
2	Cust_St_Dt	Customer_Start_Date
3	ThisColumnWillEndUpWithUnderscores	This_Column_Will_End_Up_With_Underscores
4	SalesforceTypeColumnName__c	Salesforce_Type_Column_Name
5	Record_Last_Modified	Record_Last_Modified
6	IsDeleted	Is_Deleted
7	Staging_Id	DimExampleColumnNameConversion_did
8	SourceSystem_Id	SourceSystem_Id
9	ETLBatch_Id	ETLBatch_Id
10	DataState_Id	DataState_Id

Example 6: Modified Dates

As part of the ETL build process, BI Builder runs a stored procedure to detect a column in the staging table that would indicate when each record was last updated (or created). This column is used in the generated ETL to ensure that fresh records are not overwritten with stale records.

The stored procedure is named **[bib].[FlagModifiedDateFieldInStagingTable]**.

It is currently setup to flag any column with 'Modified" or "Update" in its name and where the data type is DateTime.

If your data source for a dimension does not contain any fields that indicate when the data was last updated, the ETL will still be built and will function without the stale data checking.

If you re-run the script for Example 1, you will see in the table **[bib].[ColumnDefinitions]** that the column "Record_Last_Updated" is being flagged in the "`IsDataCaptureDateTime`" column and applied as a filter in the generated ETL stored procedures.

As shown in Example 4, a column that is flagged as being the "Modified Date" column can also be transformed in data type and still function properly as a filter in the generated ETL stored procedures. See the examples around transforming data types in this document.

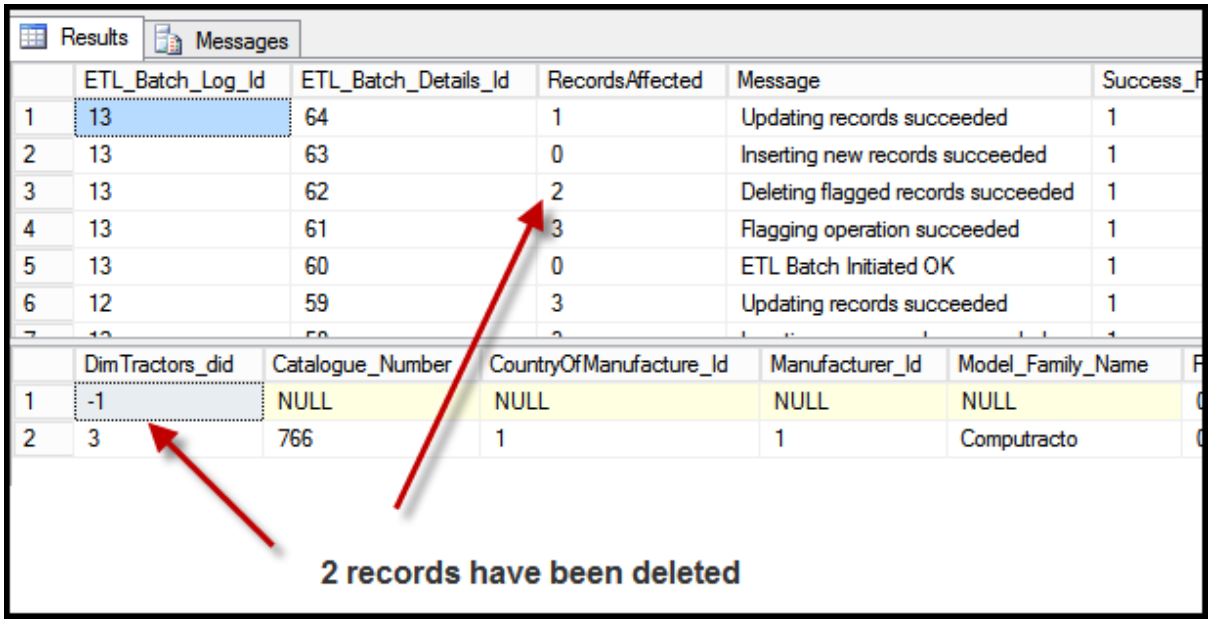
Example 7: Handling Deleted Records

We will re-run the script for Example 1 and then test deletion of a record using a test script that BI Builder provides when it generates the ETL stored procedures.

Load the SQL file "**`Example 1 A Simple Type 1 Dimension.sql`**" into a query window and execute it.

In SMSS Object Explorer window, right click on the stored procedure [Examples].[Load_DimTractors] and select "Modify". Amongst the test routines in the header you will see the following:

```
-- To test deletions
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,1,1,1,0 -- reset and run first, no log report

EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,1,0,0 -- restage the data

UPDATE   [Example_Staging].[Tractors]
    SET [Record_Deleted] = 1 WHERE Staging_Id < 3 -- flag the first 2 records as deleted

EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,1,0 -- run the load

EXECUTE   [Examples].[Load_DimTractors_ManualTesting] 0,0,0,0,1 -- check the logs to see that
records
    --were inserted and updated, then in one batch
    --there were 2 deletions and 1 update

SELECT * FROM       [Examples].[DimTractors] -- checkout the records in the dimension to ensure the
                                              --flagged record(s) have been deleted
```



	ETL_Batch_Log_Id	ETL_Batch_Details_Id	RecordsAffected	Message	Success_F
1	13	64	1	Updating records succeeded	1
2	13	63	0	Inserting new records succeeded	1
3	13	62	2	Deleting flagged records succeeded	1
4	13	61	3	Flagging operation succeeded	1
5	13	60	0	ETL Batch Initiated OK	1
6	12	59	3	Updating records succeeded	1

	DimTractors_did	Catalogue_Number	CountryOfManufacture_Id	Manufacturer_Id	Model_Family_Name	F
1	-1	NULL	NULL	NULL	NULL	
2	3	766	1	1	Computracto	

Example 8: A Table with 300 fields

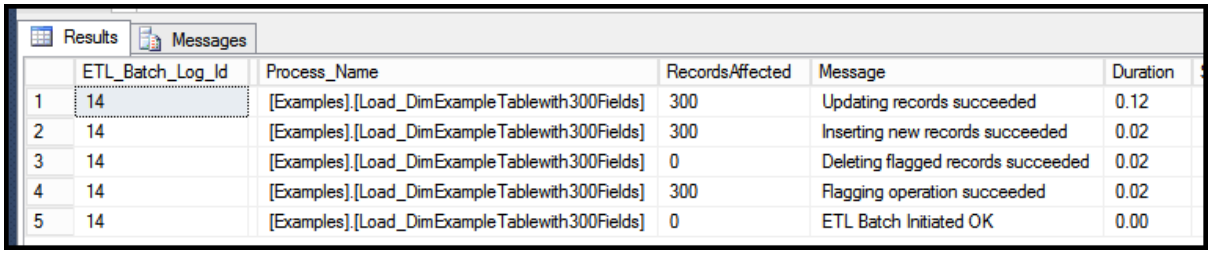
Writing ETL code for massive tables can be tedious and error prone when working by hand without tool support. One aim of BI Builder is to make working with large tables almost as easy as working with small simple tables.

This example uses a model table with 300 columns of various data types, such as XML, as a demonstration. There is a helper routine that populates this table with 300 rows – this is executed within the following script that builds the ETL.

```
-- ETL BUILD SCRIPT FOR TABLE [Models].[ExampleTablewith300Fields]
DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @StagingFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'ExampleTablewith300Fields'

-- Staging Table Details ------------
    SET @StagingFilegroupName = 'Example_Staging'
    SET @StagingSchemaName = 'Example_Staging'
    SET @StagingTableName = 'ExampleTablewith300Fields'
    SET @SlowlyChangingDimensionColumnList = NULL

-- Dimension Table Details ------------
    SET @DimSchemaName = 'Examples'
    SET @DimTableName = 'DimExampleTablewith300Fields'
    SET @DimFilegroupName  = 'Examples'
    SET @SourceSystem_Id  = 1

-- Control Flags for BI Builder Main()
    SET @SkipDataAnalysis = 1
    SET @ApplyAutoAudit = 0

    ---- 1) Control Flags for automated end-to-end build process
    SET @PauseForManualEditsToMetadata = 0
    SET @UseCurrentMetadataForBuild = 0

-- Populate the example table
EXECUTE [Examples].[GenerateTestDataForExampleTablewith300Fields]

EXECUTE[bib].[Main]
   @ModelSchemaName
  ,@ModelTableName
  ,@StagingSchemaName
  ,@StagingTableName
  ,@StagingFilegroupName
  ,@SourceSystem_Id
  ,@SlowlyChangingDimensionColumnList
  ,@DimSchemaName
  ,@DimTableName
  ,@DimFilegroupName
  ,@SkipDataAnalysis
  ,@PauseForManualEditsToMetadata
  ,@UseCurrentMetadataForBuild
  ,@ApplyAutoAudit

GO
```

Now execute the generated ETL and view the logs using the test scripts in the header of the generated stored procedure [Examples].[Load_DimExampleTablewith300Fields]:

```
-- To run process on whatever is in the staging table and show log
EXECUTE  [Examples].[Load_DimExampleTablewith300Fields_ManualTesting] 0,0,0,1,1
```



Results | Messages

	ETL_Batch_Log_Id	Process_Name	RecordsAffected	Message	Duration
1	14	[Examples].[Load_DimExampleTablewith300Fields]	300	Updating records succeeded	0.12
2	14	[Examples].[Load_DimExampleTablewith300Fields]	300	Inserting new records succeeded	0.02
3	14	[Examples].[Load_DimExampleTablewith300Fields]	0	Deleting flagged records succeeded	0.02
4	14	[Examples].[Load_DimExampleTablewith300Fields]	300	Flagging operation succeeded	0.02
5	14	[Examples].[Load_DimExampleTablewith300Fields]	0	ETL Batch Initiated OK	0.00

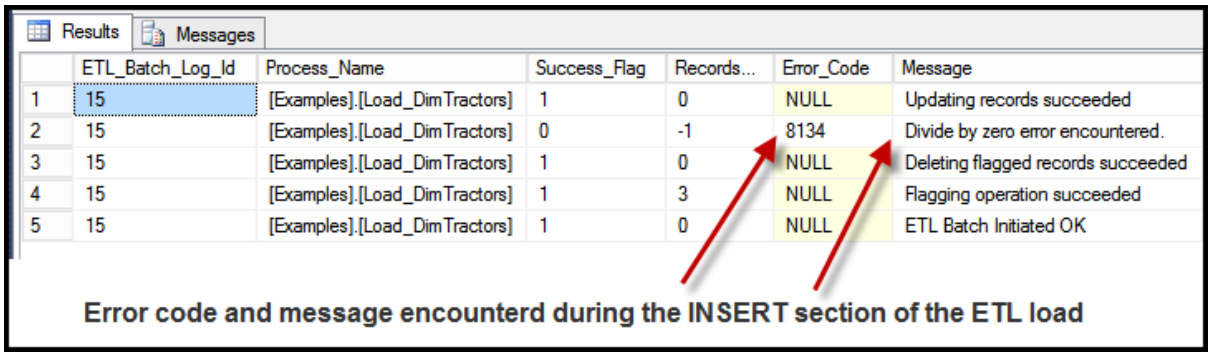
Example 9: Forcing an Error

Using a stored procedure we have already generated, **[Examples].[Load_DimTractors]**, open it up in a query window and make the following edit to force a "divide by zero" error within the INSERT section.

```
        INSERT INTO [Examples].[DimTractors](
                    [Record_Deleted]
                    , [Record_Last_Updated]
                    , [Tractor_Id]
                    , [SourceSystem_Id]
                    , [ETLBatch_Id]
                    ,[DataState_Id]
        )

        SELECT
                    [Record_Deleted]
                    , [Record_Last_Updated]
                    , 1/0 --Instead of [Tractor_Id], to force an error
                    , [SourceSystem_Id]
                    , [ETLBatch_Id]
                    , 4
        FROM
                    [Example_Staging].[Tractors]
```

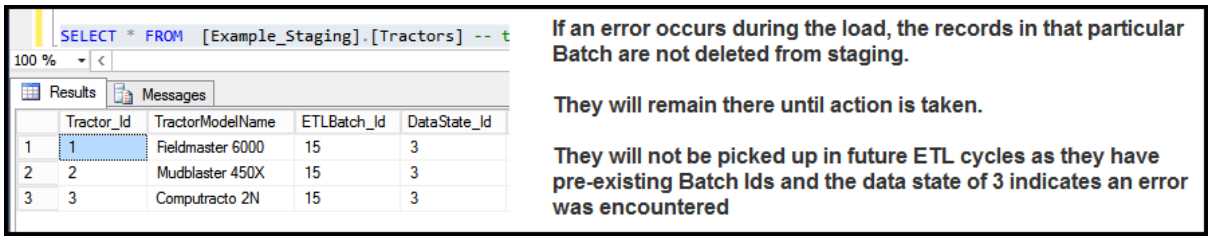
To test error handling, we will restage the data from the model table and feed it into the dimension. Then we will show the logs for this batch to verify that the error is clearly flagged.

```
-- Clear the logs, restage the test data and truncate the dimension
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 1,1,1,0,0

EXECUTE [Examples].[Load_DimTractors]

-- See the logs
EXECUTE  [Examples].[Load_DimTractors_ManualTesting] 0,0,0,0,1

SELECT * FROM  [Example_Staging].[Tractors] -- to confirm data has not been cleaned out
```



Results | Messages

	ETL_Batch_Log_Id	Process_Name	Success_Flag	Records...	Error_Code	Message
1	15	[Examples].[Load_DimTractors]	1	0	NULL	Updating records succeeded
2	15	[Examples].[Load_DimTractors]	0	-1	8134	Divide by zero error encountered.
3	15	[Examples].[Load_DimTractors]	1	0	NULL	Deleting flagged records succeeded
4	15	[Examples].[Load_DimTractors]	1	3	NULL	Flagging operation succeeded
5	15	[Examples].[Load_DimTractors]	1	0	NULL	ETL Batch Initiated OK



```
SELECT * FROM [Example_Staging].[Tractors] -- t
```

100 %

Results | Messages

	Tractor_Id	TractorModelName	ETLBatch_Id	DataState_Id
1	1	Fieldmaster 6000	15	3
2	2	Mudblaster 450X	15	3
3	3	Computracto 2N	15	3

Example 10: Working from your own data source

1) Create a table in the example database, ensure it has a primary key and populate it with a solid sample set of records. You do not need to put this table in the schema [Models] but that is recommended. This table is not your actual staging table, it is a model of your staging table that won't be modified by BI Builder.

2) Execute the following stored procedure to generate an ETL Build Script

```
EXECUTE [BIB].[Main_PrintScriptToBuildETLFromModelTable] '<YOUR_SCHEMA>', '<YOUR_TABLE>'
```

In the build script that is produced, you will enter the names and schemas of your staging and dimension tables, as well as the filegroups you want these to reside on. If those schemas and filegroups don't exist, they will be created.

Then execute the script and test the results as we have in the previous examples, using the test scripts in the header of the main stored procedure generated - [Load_<DimTableName>].

Example 11 – Pausing a build to make ad-hoc changes to the metadata

BI Builder goes through two phases when it builds ETL components for a model table.

1) Building the staging table and trying to auto-detect crucial metadata such as whether a column might indicate soft-deletion of the record or if it is a business key.
2) Building the dimension table and related stored procedures.

If you are going to strike a common column in all your source tables that needs special treatment, you are best off modifying the auto-detection routines to encode your detection rules. For example, you might have a column in every table named "X" and if the value = 1 then the record is to be considered deleted. You can get BI Builder to treat that as a deletion flag column by detecting it in the stored procedure `[BIB].[FlagDeletedBitFieldInStagingTable]` .

But we will use an example of a one-off where it is faster for you to tweak the build script than edit the BI Builder stored procedures.

In one source table, dates are encoded as text fields. In your dimension, you want these to be expressed as DateTime fields so you can easily connect them to your calendar dimension.

Execute this stored procedure to create and ETL Build Script for the table **[Models].[ExampleOfPausingETLBuilds]**

```
EXECUTE [BIB].[Main_PrintScriptToBuildETLFromModelTable]
                @ModelSchemaName = 'Models'
              , @ModelTableName  = 'ExampleOfPausingETLBuilds'
```

Notice that in the script produced, there is a section that lets you comment/uncomment parameters that let you pause the build and also to restart it at the point where the dimension and stored procedures are created.

```
-- ETL BUILD SCRIPT FOR TABLE [Models].[ExampleOfPausingETLBuilds]
.. details hidden …
-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'ExampleOfPausingETLBuilds'
.. details hidden …
    -- CHOOSE ONE OF THE FOLLOWING THREE OPTION SETS, DEPENDING ON HOW YOU WANT TO BUILD THE ETL
    ------------------------------------------------------------------------------------------

    ---- 1) Control Flags for automated end-to-end build process
    SET @PauseForManualEditsToMetadata = 0
    SET @UseCurrentMetadataForBuild = 0

    ---- 2) Control Flags to pause build process in order to manually edit the metadata from a script that is
generated in the build print output
    --SET @PauseForManualEditsToMetadata = 1
    --SET @UseCurrentMetadataForBuild = 0

    -- 3) Control Flags to build from your manual changes to the metadata
    -- This configuration skips the metadata creation process that would otherwise overwrite your changes
    --SET @PauseForManualEditsToMetadata = 0
    --SET @UseCurrentMetadataForBuild = 1

EXECUTE[bib].[Main]
.. details hidden …
```

In this example, we want to first use option 2 to pause the build and then option 3 to finish the build.

Here is the script we will run first:

```
-- Example 11 - Pausing an ETL Build to edit metadata
```

```
-- ETL BUILD SCRIPT FOR TABLE [Models].[ExampleOfPausingETLBuilds]
-------------------------------------------------------------------------------
DECLARE @ModelSchemaName nvarchar(255)
DECLARE @ModelTableName nvarchar(255)
DECLARE @StagingSchemaName nvarchar(255)
DECLARE @StagingTableName nvarchar(255)
DECLARE @StagingFilegroupName nvarchar(255)
DECLARE @SourceSystem_Id int
DECLARE @SlowlyChangingDimensionColumnList nvarchar(2048)
DECLARE @DimSchemaName nvarchar(255)
DECLARE @DimTableName nvarchar(255)
DECLARE @DimFilegroupName nvarchar(255)
DECLARE @SkipDataAnalysis bit
DECLARE @PauseForManualEditsToMetadata bit
DECLARE @UseCurrentMetadataForBuild bit
DECLARE @ApplyAutoAudit bit

-- Model Table Details -----
    SET @ModelSchemaName = 'Models'
    SET @ModelTableName = 'ExampleOfPausingETLBuilds'

-- Staging Table Details ------------
    SET @StagingFilegroupName = 'Example_Staging'
    SET @StagingSchemaName = 'Example_Staging'
    SET @StagingTableName = 'ExampleOfPausingETLBuilds'
    SET @SlowlyChangingDimensionColumnList = NULL

-- Dimension Table Details ------------
    SET @DimFilegroupName  = 'Examples'
    SET @DimSchemaName = 'Examples'
    SET @DimTableName = 'DimExampleOfPausingETLBuilds'
    SET @SourceSystem_Id  = 1

-- Control Flags for BI Builder Main()
    SET @SkipDataAnalysis = 0
    SET @ApplyAutoAudit = 0

    -- CHOOSE ONE OF THE FOLLOWING THREE OPTION SETS, DEPENDING ON HOW YOU WANT TO BUILD THE ETL
    ---------------------------------------------------------------------------------------------

    ---- 1) Control Flags for automated end-to-end build process
    --SET @PauseForManualEditsToMetadata = 0
    --SET @UseCurrentMetadataForBuild = 0

    ---- 2) Control Flags to pause build process in order to manually edit the ---- metadata from a
script that is generated in the build print output
    SET @PauseForManualEditsToMetadata = 1
    SET @UseCurrentMetadataForBuild = 0

    -- 3) Control Flags to build from your manual changes to the metadata
    -- This configuration skips the metadata creation process that would otherwise overwrite your
changes
    --SET @PauseForManualEditsToMetadata = 0
    --SET @UseCurrentMetadataForBuild = 1

EXECUTE[bib].[Main]
   @ModelSchemaName
  ,@ModelTableName
  ,@StagingSchemaName
  ,@StagingTableName
  ,@StagingFilegroupName
  ,@SourceSystem_Id
  ,@SlowlyChangingDimensionColumnList
  ,@DimSchemaName
  ,@DimTableName
  ,@DimFilegroupName
  ,@SkipDataAnalysis
  ,@PauseForManualEditsToMetadata
  ,@UseCurrentMetadataForBuild
  ,@ApplyAutoAudit

GO
```

The execution provides the usual BI Builder message log but towards the end of that you will see a prepared script you can copy into a new window in order to alter the metadata.

This is the first fragment of that generated script:

```
        -- ###########################################################################################
        -- SCRIPT TO MANUALLY EDIT THE METADATA FOR THE STAGING TABLE
        -- [Example_Staging].[ExampleOfPausingETLBuilds]

        -- You edit this to suit and then continue the ETL build using the [bib].[Main] stored procedure

        -- You can protect your changes by running [bib].[Main] with the parameter
        -- @UseCurrentMetadataForBuild = 1

        -- (When UseCurrentMetadataForBuild = 0, it will overwrite any metadata changes made manually)

        -- ###########################################################################################

-- UPDATE METADATA FOR STAGING COLUMN:
--       Cust_Id

UPDATE [bib].[ColumnDefinitions]
   SET
       [DataType] = 'int'
      ,[CharacterMaximumLength] = 4
      ,[DimensionColumnName] = 'Cust_Id'
      ,[DimensionDataType] = 'int'
          ,[DimensionDataTypeLength] =  4
      ,[CustomTypeConversionExpression] =   NULL
      ,[IncludeInDWH] =   1
      ,[UseColumnForSCD] =   0
      ,[IsBusinessKey] =  0
      ,[IsDataCaptureDateTime] =   0
      ,[IsDeletedFlag] =  0
FROM
            [bib].[ColumnDefinitions]
            WHERE
                  ColumnDefinition_Id = 729
```

The change we want to make to the ETL is this – convert the [Start_Date] column from a string date to a DateTime Date.
This means we want to pop in a value for the [CustomTypeConversionExpression] column.

The script that would do that is this – execute it:

```
-- UPDATE METADATA FOR STAGING COLUMN:
--       StartDate

UPDATE [bib].[ColumnDefinitions]
   SET
       ,[DimensionDataType] = 'DateTime' <- we overwrote the "nvarchar"
       ,[CustomTypeConversionExpression] =   'CAST(<COLUMN> as DateTime)'
FROM
           [bib].[ColumnDefinitions]
           WHERE
               ColumnDefinition_Id = 730

(NOTE "ColumnDefinition_Id = 730" is specific to the author's build, you must
collect the script from the output of [bib].[Main] in the earlier part of this
example)
```

Then you can resume the build by altering your ETL Build script to comment out one set of control flags and uncomment the other:

```
-- CHOOSE ONE OF THE FOLLOWING THREE OPTION SETS, DEPENDING ON HOW YOU WANT TO BUILD THE ETL
-------------------------------------------------------------------------------------------

---- 1) Control Flags for automated end-to-end build process
--SET @PauseForManualEditsToMetadata = 0
--SET @UseCurrentMetadataForBuild = 0

-- 2) Control Flags to pause build process in order to manually edit the metadata
--SET @PauseForManualEditsToMetadata = 1
--SET @UseCurrentMetadataForBuild = 0

-- 3) Control Flags to build from your manual changes to the metadata
-- This configuration skips the metadata creation process that would otherwise overwrite your changes
SET @PauseForManualEditsToMetadata = 0
SET @UseCurrentMetadataForBuild = 1
```

Comment this out to avoid overwriting your changes to the metadata

Uncomment this to pick up the build at the point where it creates the dimension and stored procedures

Run the ETL Build script again and you will finish the build.

We can see that our generated dimension table has a DateTime column in place of the staging table's nvarchar column.

	Column Name	Data Type	Allow Nulls
🔑	DimExampleOfPausingETLBuilds_did	bigint	☐
	Cust_Id	int	☐
	Cust_Type_Code	int	☑
	Is_Deleted	bit	☐
	PrimaryKey_Id	int	☐
	Start_Date	datetime	☑
	SourceSystem_Id	int	☐
	ETLBatch_Id	int	☑
	DataState_Id	int	☑
			☐

Converted to DateTime from nvarchar through editing the metadata

Also in the generated stored procedure **[Load_DimExampleOfPausingBuilds]** we see the conversion:

The conversion has been built into the ETL Stored Procedure [Load_DimExampleOfPausingETLBuilds]

```
UPDATE dest
   SET
       dest.[Cust_Id] = src.[Cust_Id]
     , dest.[Cust_Type_Code] = src.[CustTypeCode]
     , dest.[Is_Deleted] = src.[IsDeleted]
     , dest.[Start_Date] = CAST(src.[StartDate] as DateTime)
     , dest.[ETLBatch_Id] = src.[ETLBatch_Id]
       , [DataState_Id] = 5
  FROM
       [Examples].[DimExampleOfPausingETLBuilds] dest
  INNER JOIN  [Example_Staging].[ExampleOfPausingETLBuilds] src
  ON
  dest.[PrimaryKey_Id] =  src.[PrimaryKey_Id]
       AND dest.[SourceSystem_Id] =  src.[SourceSystem_Id]
    WHERE
    src.[ETLBatch_Id] = @Batch_ID
    AND
    src.DataState_Id = 1
    AND
    src.[IsDeleted] = 0
```

For further information about BI Builder and its usage

Please contact the author Patrick Connors via pconnors@curiousdevelopments.net

Installing BI Builder into your development database

It is recommended before installing into a real development data warehouse that you checkout BI Builder through the example database described in an early section of this document.

BI Builder installs into development databases and operates on the database that it is installed into.

Any objects in the [BIB] schema are purely for development and have no role in a production system.

There are however a small number objects in the [ETL] schema that your generated tables and stored procedures will be dependent on and there is a script later in this document for doing the production installation.

It is recommended you install AutoAudit into your data warehouse before BI Builder but this is not essential. BI Builder can make use of AutoAudit if it is available. Details for obtaining this script can be found in the next section where it describes setup of the example database.

To install BI Builder into your MSSQL database, load and execute the script named

`BIBuilder <version> Development Database Setup Script.sql`

in a query window connected to your development data warehouse.

Components to be installed into Production

The objects in the BIB schema do not go into production or testing systems, they are for development only. The objects you produce with BI Builder do go into production, along with a few objects those generated components will be dependent on.

These dependencies are the objects in the [ETL] schema. The setup script for these objects is named "**BI Builder Production Object Setup Script.sql**".

In order to run your ETL in production, create a stored procedure and call it [ETL].[ETLDriver].

In that stored procedure, you would execute all the [Load_DimX] stored procedures that drive the loading for each dimension table.

You would then schedule the [ETLDriver] routine to run the ETL as a SQL Agent job at the desired frequency that fits the rate at which data updates are flowing in from outside the data warehouse.

Guide to Modifying BI Builder

BI Builder is composed of a driver routine [bib].[Main] that constructs the database objects by calling out to various stored procedures in a definite order.

Currently there are 21 major steps and you will find these in [BIB].[Main].

Many of these stored procedures call out to scalar functions in order to construct parts of SQL statements such as lists of key columns or lists of non-nullable columns or JOIN clauses.

The individual stored procedures and scalar functions are documented in the following appendix however a few key ones will be discussed here. These procedures generally have a parameter option to view the generated script without executing it – this can be handy if you are debugging changes you have made to the scripts.

Modifying Staging Table Construction

The stored procedure `[BIB].[CreateStagingTableFromModel]` formulates the script to create the staging table from your model table.

The columns for ETL management are added in `[BIB].[AddETLColumnsToStagingTable]` if you want to change the columns used to manage your ETL.

Modifying Dimension Table Construction

The stored procedure `[BIB].[GenerateDimensionTableScript]` formulates the script to create the dimension table from your staging table. It also creates the unique indexes from your business keys.

Modifying ETL Stored Procedure Construction

The stored procedures generated to drive the ETL are produced by the stored procedure `[BIB].[GenerateDimLoadETLStoredProcedures]`. By examining this routine, you will see the BIB objects in which you may wish to make changes.

Applying AutoAudit to your generated Dimension Table

When you run the main procedure, there is a parameter named @ApplyAutoAudit.

Provided you have AutoAudit installed in your data warehouse, it will setup auditing on the generated dimension table.

This is a good thing to do if you consider the data in a dimension qualifies as Master Data and needs a good level of auditing and potentially easy rollback.

The only problem would be a performance hit loading the dimension – a lot of trigger action would be generated if you are loading batches of 10,000 records at a time – so it is recommended that you avoid applying it to very large tables that undergo a lot of daily change, such as transaction tables.

APPENDIX: Documentation of Stored procedures and Scalar Functions

Main

Stored Procedure [bib].[Main]

This is the routine that drives the entire ETL creation process.

To generate a script that helps you build an ETL from a Model Table, run this routine:

```
EXECUTE [bib].[Main_PrintScriptToBuildETLFromModelTable] '<ModelSchema>', '<ModelTableName>'
```

AddETLColumnsToStagingTable

Stored Procedure [bib].[AddETLColumnsToStagingTable]

The staging table is augmented automatically during the build with columns that help in managing the ETL process.

These are

- Staging_Id - a surrogate key used purely in staging 2. SourceSystem_Id - used to identify the source system
- ETLBatch_Id - populated with the Btach Id during a load
- DataState_Id - indicates the status of the data

The data states are:

- 1 Staging - Flagged for load into dimension within batch
- 2 Staging - Loaded successfully into dimension within batch, ready for deletion
- 3 Staging - Error occurred loading records into dimension within batch
- 4 Dimension - Partial load of non-nullable columns (Part 1 of 2) - Not ready for reporting
- 5 Dimension - Load of all batched records complete (Part 2 of 2) - Ready for reporting
- 6 Dimension - Frozen, cannot be updated (Can only be set manually)

AnalyseSourceTable

Stored Procedure [bib].[AnalyseSourceTable]

Called within the ETL build process (if option selected), this routine drives all the stored procedures that run analytics on the staging table data. The results of the analytics are stored in their own columns of the **[bib].[ColumnDefinitions]** table.

If you create new analytic routines, add them in here.

ColumnDefinitions_AddColumnsFromTable

Stored Procedure [bib].[ColumnDefinitions_AddColumnsFromTable]

Captures the metadata for the columns in the staging table that will be used to generate the dimensions and ETL stored procedures.

ColumnDefinitions_AddETLColumnsFromTable

Stored Procedure [bib].[ColumnDefinitions_AddETLColumnsFromTable]

Called from within the ETL build process, this routine captures the metadata for the specialised ETL management fields that have been automatically added to the staging table.

ConvertBusinessKeysToUniqueIndexOnStagingTable

Stored Procedure [bib].[ConvertBusinessKeysToUniqueIndexOnStagingTable]

Called from within the ETL build process, this routine takes those fields flagged as business keys earlier on and uses them to make a unique index to enforce no duplicates.

The SourceSysytem_Id field will also be part of that unique index.

The index is also applied to the generated dimension table when it gets built later on.

CreateColumnNamesForDimension

Stored Procedure [bib].[CreateColumnNamesForDimension]

Called from within the ETL build process, this routine uses a function to determine the best names for the generated dimension table columns.

The logic for the name transformation rules are contained within bib.MakeDimensionColumnNameFromSourceColumn.

You would use this to make the column names that report users will ultimately see as readable as they can be.

For example, your data source might use abbreviations for common terms such as "cust" for "Customer" or "prd" for "Product"

If the function bib.MakeDimensionColumnNameFromSourceColumn, you can set a rule that expands this out to "Customer".

The function also introduces "_" to PascalCase field names, converting a field like "CustOrder" to "Customer_Order" in the generated dimension.

When that field is imported into Analysis Services, it will read to the end consumer as "Customer Order" without you having to edit anything.

Do note that the routine does not meddle with fields that end in "Id" as these are most likely keys and there are benefits in keeping these names consistent with the source. These fields are also unlikely to appear in user reports and so there would be little to gain by transforming them.

CreateDataTypesForDimension

Stored Procedure [bib].[CreateDataTypesForDimension]

Called within the ETL build process, this routine examines the data types used in the staging table and determines the data types to be used in the dimension table.

Thescalar functions that specify the rules are:
bib.MakeDimensionDataTypeFromSourceColumn
bib.MakeDimensionDataTypeLengthFromSourceColumn

If you want to create your own rules, edit those scalar functions above.

CreateStagingTableFromModel

Stored Procedure [bib].[CreateStagingTableFromModel]

Called from within the ETL build process, this routine creates and executes a script to build the staging table from your model table.

DeletePrimaryKeyFromStagingTable

Stored Procedure [bib].[DeletePrimaryKeyFromStagingTable]

Called from within the ETL build process, this routine removes the primary key from the staging table and replaces it with a surrogate key.

The original primary key has already been noted and flagged as a "business Key" to be used in the logic of the generated ETL stored procedures.

ExcludeColumnsFromDWHBasedOnRules

Stored Procedure [bib].[ExcludeColumnsFromDWHBasedOnRules]

Called automatically from within the ETL build process, this routine applies rules you specify to exclude staging columns from being represented within the generated dimension table.

All the tables in your data source might have a commonly named field that you know to be useless from a data warehouse perspective. This lets you set a rule to exclude themwithout you having to think about it each time to build a new dimension.

There is a commented-out example rule that would have you exclude any column where the sample data is 100% NULL.

FlagBusinessKeysFromStagingTable

Stored Procedure [bib].[FlagBusinessKeysFromStagingTable]

The model table you specify to build the tables and ETL stored procedures from will have a primary key (and possibly a composite key).

This routine flags those fields in the metadata in order to use them as business keys in the generated ETL routines and so they form part of the unique indexes on both the generated staging and dimension tables as a way of defending against storing duplicate records.

FlagDeletedBitFieldInStagingTable

Stored Procedure [bib].[FlagDeletedBitFieldInStagingTable]

This routine is called within an ETL build in order to flag a column that indicates whether or not the record is deleted.

If you data sources conform to some naming convention around columns used for soft deletes, you can ensure that this routine below detects them and flags them.

It is currently dependent on the "deleted" field being a BIT field. This field is encoded into the ELT stored procedures to both delete existing expired records from the dimension (if desired) and also to ensure deleted records are not entered into the dimension.

If there is no column to indicate if a record has been deleted in the source system, the generated ETL will still function.

FlagModifiedDateFieldInStagingTable

Stored Procedure [bib].[FlagModifiedDateFieldInStagingTable]

This routine is called within an ETL build in order to flag a column that indicates when the record was either created or last modified.

If you data sources conform to some naming convention around columns used for Modified Dates, you can ensure that this routine below detects them and flags them.

If there is no column to indicate when a record has been created or modified in the source system, the generated ETL will still function but you may lose the capacity to tell if you are overwriting current information with stale records that have somehow been fed into the staging layer.

FlagSCDKeyColumnsFromStagingTable

Stored Procedure [bib].[FlagSCDKeyColumnsFromStagingTable]

This routine flags the staging columns that have been chosen as columns to track within a Type 2 Slowly Changing Dimension.

Changes in these columns trigger the creation of a new active record in the generated dimension.

To create a SCD Type 2 Dimension and ETL, you supply a comma delimited list of staging column names in the parameters for [bib].[Main]. There is a placeholder in the script generated by [bib].[Main_PrintScriptToBuildETLFromModelTable] for listing the SCD columns.

GenerateDebuggingScriptForDimension

Scalar Function [bib].[GenerateDebuggingScriptForDimension]

Generates the script you can use post-ETL build in order to test the generated objects.

This script is also appended to the header of the main ETL procedure generated

GenerateDimensionTableScript

Stored Procedure [bib].[GenerateDimensionTableScript]

This routine creates and executes a script to build the dimension table.

It bases the table on the staging table that was in turn created from the model table supplied as a parameter to the build.

It uses the dimension column names and data types from the bib.ColumnDefinitions table - which are automatically derived but can be tweaked manually if you want.

It recreates the DEFAULT CONSTRAINTS and CHECK CONSTRAINTS from staging but not references to foreign keys.

It creates a surrogate key also.

If you specify to have auditing applied through the build process for this dimension, that will be applied later in the build after the table has been created.

GenerateDimLoadETLStoredProcedures

Stored Procedure [bib].[GenerateDimLoadETLStoredProcedures]

This routine creates the set of stored procedures that manage the dimension you are building.

One of the created routines drives all the others. It is the one named [Load_<DIMENSION_TABLE_NAME>].

If you look into that procedure after the build, you will see a header that contains all the SQL you need to easily test the newly created routines using the data from the model table you started with.

GenerateMetadataEditingScript

Scalar Function [bib].[GenerateMetadataEditingScript]

You have the option of running the Main routine so it pauses after it has collected the metadata it will use to build the dimension table and ETL stored procedures.

If you choose that option, part of the output will be a script that makes editing the metadata easier.

This is the routine that generates that script.

LongPrint

Stored Procedure [bib].[LongPrint]

Prints out text of any length into the SMSS output window without having to worry about long text being truncated

This routine was created by Adam Anderson.

Refer to the original posting:

http://blog.falafel.com/t-sql-exceeding-the-8000-byte-limit-of-the-print-statement/

ModelTableHasPrimaryKey

Scalar Function [bib].[ModelTableHasPrimaryKey]

Returns 1 if the model table you are building from has a primary key.

PrebuildCheckingForStagingTable

Stored Procedure [bib].[PrebuildCheckingForStagingTable]

This routine is called within the build to verify that the staging table conforms to BI Builder's requirements.

You do have the option of starting with an existing staging table (rather than a model table from which a staging table is created). If you are doing this and you wnat to build in checks that suit your changes to BI Builder, add those checks here.

For example, you might enforce that staging tables all have a standard column name for a column that indicates the record is deleted or that certain columns have certain CHECK CONSTRAINTS. You would run those checks in here

RemoveTableMetadataFromBIBuilder

Stored Procedure [bib].[RemoveTableMetadataFromBIBuilder]

Removes all metadata concerning an individual staging table from BI Builder. Any subsequent ETL builds for that staging table will start from scratch.

RepositionColumnsForDimension

Stored Procedure [bib].[RepositionColumnsForDimension]

Adjusts the ordinal position of the columns in the generated dimension table. It pushes the ETL-process columns to the end, the business keys to the front and orders the rest in between alphabetically

Restart_BIB_ETL_Build_Progress

Stored Procedure [bib].[Restart_BIB_ETL_Build_Progress]

Restarts the logging for an ETL build based on a particular staging table.

SetCustomTypeConversionExpressionThroughRules

Stored Procedure [bib].[SetCustomTypeConversionExpressionThroughRules]

Sometimes the staging table data will come in with data expressed in a type you want to change.

An example might be a UNIX style data field in which the value is expressed as the number of seconds since 1 Jan 1970. In your DWH, you want this to be expressed as a DateTime but in staging the data comes through as an integer.

In order to achieve that conversion of data types in the the generated ETL stored procedures, you specify a custom type conversion.

In this routine now, there is an example conversion for the scenario described above. The assumption behind the example is that you will deal with many tables from that source that all use that date format and are detectable through name/data-type combinations -

therefore it is built into the stored procedures that govern the ETL build and you dont have to think about it too much.

If you only strike one table that needs a particular transformation, you can optionally just edit the [bib].[ColumnDefinitions] table directly when building that particular dimension table and related ETL.

One of the options when you run [bib].[Main] is to pause to edit the collected metadata. When you select that option, you are provided with an SQL script that makes this process simple.

SetSurrogateKeyNameForDimension

Stored Procedure [bib].[SetSurrogateKeyNameForDimension]

Called within the build process, this routine applies a standardised naming convention to the surrogate key.

By default, the surrogate key is named [_did].

Did is used instead of Id to differentiate dimension keys from business keys, Did stands for Dimension Id.

If you wish to change that, you can edit this routine to suit your naming standards.

TableDefinitions_AddTable

Stored Procedure [bib].[TableDefinitions_AddTable]

Called within the ETL build process to store metadata about the staging table (not the columns).

TableIsLockedToOverwriting

Scalar Function [bib].[TableIsLockedToOverwriting]

Returns "1" if the table is locked such that you cannot regenerate the ETL and dimensions from a particular staging table.

Update_BIB_ETL_Build_Progress

Stored Procedure [bib].[Update_BIB_ETL_Build_Progress]

Used within the ETL build process to track what steps have been performed successfully.

You will never need to call this routine manually.

GeneratePrimaryKeyDROPClause

Scalar Function [bib].[GeneratePrimaryKeyDROPClause]

Generates the SQL that is executed within the ETL build process to drop the primary key on the staging table.

AnalyseTableForDateColumnsStoredAsStringColumn

Stored Procedure [bib].[AnalyseTableForDateColumnsStoredAsStringColumn]

An example of analysis you can run and feed back into the metadata.

This routine looks into the values held in string columns to determine if the values are all dates and best stord in the generated dimension as [DateTime]s.

Ideally you would be able to get your data types fully sorted out in your model table but it is possible you are constrained by legacy ETL systems delivering data to your data warehouse.

AnalyseTableForDistinctValueCountsPerColumn

Stored Procedure [bib].[AnalyseTableForDistinctValueCountsPerColumn]

An example of analysis you can run and feed back into the metadata.

This routine looks at how many distinct values you have amongst each column of the staging table. The results are stored in [bib].[ColumnDefinitions]

AnalyseTableForFloatColumnsStoredAsStringColumn

Stored Procedure [bib].[AnalyseTableForFloatColumnsStoredAsStringColumn]

An example of analysis you can run and feed back into the metadata prior to building the dimensions and ETL stored procedures.

This routine looks into the values held in string columns to determine if the values are all floating point and best stored in the generated dimension as a floating point type.

Ideally you would be able to get your data types fully sorted out in your model table but it is possible you are constrained by legacy ETL systems delivering data to your data warehouse.

AnalyseTableForIntColumnsStoredAsStringColumns

Stored Procedure [bib].[AnalyseTableForIntColumnsStoredAsStringColumns]

An example of analysis you can run and feed back into the metadata prior to building the dimensions and ETL stored procedures.

This routine looks into the values held in string columns to determine if the values are all integers and best stord in the generated dimension as [int]s.

Ideally you would be able to get your data types fully sorted out in your model table but it is possible you are constrained by legacy ETL systems delivering data to your data warehouse.

AnalyseTableForIntColumnStoredAsFloatColumn

Stored Procedure [bib].[AnalyseTableForIntColumnStoredAsFloatColumn]

An example of analysis you can run and feed back into the metadata.

This routine looks into the values held in floating point columns to determine if the values are all integers and best stord in the generated dimension as [int]s.

Ideally you would be able to get your data types fully sorted out in your model table but it is possible you are constrained by legacy ETL systems delivering data to your data warehouse.

AnalyseTableForNULLColumns

Stored Procedure [bib].[AnalyseTableForNULLColumns]

An example of analysis you can run and feedback into the metadata.

This routine looks into the values held in nullable columns to determine what percentage of the values are NULLs.

If the column is rarely used, it might suit your requirements to exclude it from being part of the generated dimension.

FileGroupThatThisTableIsIn

Scalar Function [bib].[FileGroupThatThisTableIsIn]

Returns the name of the filegroup that the table resides in.

MakeDimensionColumnNameFromSourceColumn

Scalar Function [bib].[MakeDimensionColumnNameFromSourceColumn]

This function is where you define the rules you require to automatically create dimension column names from staging column names.

The current implementation shows a series of examples.

MakeDimensionDataTypeFromSourceColumn

Scalar Function [bib].[MakeDimensionDataTypeFromSourceColumn]

Here is where you apply any type transformations to the column in the dimension Ideally, you should get your types sorted out in the initial staging table so you can be alerted to violated type assumptions when loading the staging table from source rather than breaking the dimension load later on If you sort these types in the staging table before you build the dimensions using BI Builder the types you define in staging will flow through the the definition of the dimension table

However, there might be some restrictions on the ETL process loading you dimension tables that restrict the types you can use to non-ideal types, in which case you would define your preferred types in the ColumnDefinitions.DimensionDataType field before building the dimension in BI Builder.

for example, you might want to convert all varchars to nvarchars because you know an upcoming data source will arrive with strings expressed using unicode character sets

MakeDimensionDataTypeLengthFromSourceColumn

Scalar Function [bib].[MakeDimensionDataTypeLengthFromSourceColumn]

Here is where you apply any type length transformations to the column in the dimension Ideally, you should get your types sorted out in the initial staging table so you can be alerted to violated type assumptions when loading the staging table from source rather than breaking the dimension load later on If you sort these types in the staging table before you build the dimensions using BI Builder the types you define in staging will flow through the the definition of the dimension table

However, there might be some restrictions on the ETL process loading you dimension tables that restrict the types you can use to non-ideal types, in which case you would define your preferred types in the ColumnDefinitions.DimensionDataType field.

for example, you might want to standardise the length of strings so you are not dealing with hundreds of custom lengths You might also want to pad out string lengths if you suspect that minor changes in source systems (such as the expansion of a First_Name field) might break loads even though the actual change is trivial in terms of business rules

You will notice that the routine pads out string-type fields to allow for downstream augmentation of text. For example, imagine you have a char(1) field in the staging table for "Gender" and it has values that can only be "m" or "f" or NULL. In the DWH reports, you might want to display text instead that idicates to the report user what NULL actually means - it could be something like "Not Available" or "N/A" indicating that the person has a Gender but we do not know what it is.

By padding the string lengths out, we can hopefully avoid a lot of "binary truncation" errors in the ETL, errors that may well have no significance in terms of reporting requirements.

ForceSchemaCreation

Stored Procedure [bib].[ForceSchemaCreation]

When you build the tables and ETL routines from a Model Table, you can specify what SCHEMAs the staging and dimension tables will be created within.

If the SCHEMA does not exist, this routine is called to create it.

CountDelimitersInString

Scalar Function [bib].[CountDelimitersInString]

Returns the number of delimiters detected in a string

ExtractNthTextSection

Scalar Function [bib].[ExtractNthTextSection]

This function returns a block of text between the Nth and N+1 delimiter.

FirstSCDColumnName

Scalar Function [bib].[FirstSCDColumnName]

Used to generate the manual testing routines, this function returns the first column specifed as an SCD type 2 column for which changes will trigger new rows in the dimension.

ModifiedDateColumnName

Scalar Function [bib].[ModifiedDateColumnName]

Returns the name of the column that has been flagged (either automatically or manually) as the column indicating the last modified data of the record.

AddSCDColumnsToDimensionTable

Stored Procedure [bib].[AddSCDColumnsToDimensionTable]

Called within the ETL build process to support Slowly Changing Dimension Type 2 functionality.

Certain columns are added to the generated dimension table. These columns store the effective start and end dates for each record and whether or not the record is active or superceeded.

ForceFileGroupCreation

Stored Procedure [bib].[ForceFileGroupCreation]

When you build the tables and ETL routines from a Model Table, you can specify what FILEGROUPs the staging and dimension tables will be created within.

If the FILEGROUP does not exist, this routine is called to create it.

By default, it creates FILEs in the default data directory. If you want to override that behaviour, you can alter this procedure.

TableIsSCD

Scalar Function [bib].[TableIsSCD]

Returns 1 if the staging table metadata indicates it is to be a slowly changing dimension.

DefaultDateformatFromSettings

Scalar Function [bib].[DefaultDateformatFromSettings]

The default Date Format used for ETL loading is defined in the bib.BIB_Settings table. This function returns the value specified there.

By default the value is "dmy" (Australian standard) but you can set it to your local value. You can also edit the generated ETL stored procedures to use a custom date format for a particular dimension

GenerateDefaultValuesProcForUnknownDimensionKey

Scalar Function [bib].[GenerateDefaultValuesProcForUnknownDimensionKey]

Generates the SQL that inserts a single row of default values into the dimension table

GenerateHandleDeletedRecordsSQL

Scalar Function [bib].[GenerateHandleDeletedRecordsSQL]

Generates the SQL for the stored procedure that will handle the deletion of records from the dimension

GenerateInsertOfNewRecordsSQLForDimensionLoad

Scalar Function [bib].[GenerateInsertOfNewRecordsSQLForDimensionLoad]

Generates the SQL for loading new records into the dimension

GenerateManualLoadTestingProcedure

Scalar Function [bib].[GenerateManualLoadTestingProcedure]

When you generate the stored procedure that drives the ETL, the header contains a set of testing routines.

This is the procedure that generates the stored procedure underlying those test routines.

GenerateSCDCurrentRecordClauseSQL

Scalar Function [bib].[GenerateSCDCurrentRecordClauseSQL]

Generates the SQL that handles the SCD functionality within the generated dimension loading ETL stored prodcedures.

GenerateStagingLoadCandidateFlaggingRoutine

Scalar Function [bib].[GenerateStagingLoadCandidateFlaggingRoutine]

Generates the SQL for the stored procedure that flags staged records for loading in the current ETL batch.

GenerateUpdateSQLForDimensionLoad

Scalar Function [bib].[GenerateUpdateSQLForDimensionLoad]

Returns the SQL used in the update clause of the generated ETL stored procedure to load data from staging to the related dimension

DetectIdentityColumnInStagingTable

Stored Procedure [bib].[DetectIdentityColumnInStagingTable]

A check routine run from within the ETL build process to verify there is no IDENTITY column other than Staging_Id on the staging table.

UnderscorePascal

Scalar Function [bib].[UnderscorePascal]

This routine is used to make column names more readable when they are surfaced in Analysis Services reports.

When a column name is PascalCased - such as "CustomerOrder", this function returns "Customer_Order". Analysis Services will display the "_" as a space.

CreateFilegroupWithFile

Stored Procedure [bib].[CreateFilegroupWithFile]

Called automatically within the ETL build process to create any FILEGROUPS specified in the parameters of [bib].[Main]

CommaListOfDimensionColumns

Scalar Function [bib].[CommaListOfDimensionColumns] Returns a comma-delimited list of all columns in the dimension related to the staging table specified in the parameters.

DefaultValueForUnknownDimensionColumn

Scalar Function [bib].[DefaultValueForUnknownDimensionColumn]

In this routine, you apply the rules as to what values you will supply for the default dimension record, the one used in fact tables when there is no key into the dimension

GenerateInsertJOINClauseSQL

Scalar Function [bib].[GenerateInsertJOINClauseSQL]

Generates the SQL for the join clause on the insert statement within the Dimension load

CommaValuesListOfNonNullableStagingColumns

Scalar Function [bib].[CommaValuesListOfNonNullableStagingColumns]

Returns a comma-delimited list of either fields or conversion statements that is used to create an INSERT statement in the generated ETL stored procedures.

This list is used for inserting new records into the dimension and in the process BI Builder currently uses, it only lists the non-nullable fields.

All the other fields get updated in the subsequent action within the generated ETL stored procedure.

GenerateDeletedSafetyCatchForJOINClauseSQL

Scalar Function [bib].[GenerateDeletedSafetyCatchForJOINClauseSQL]

Generates the SQL that checks records being loaded into the dimension are not deleted - provided there is a column flagged as indicating soft deletion.

GenerateDataCopyFromModelToStagingForManualTesting

Scalar Function [bib].[GenerateDataCopyFromModelToStagingForManualTesting]

Returns the SQL used to copy the data from yor model table to the generated staging table.

CommaListOfBusinessKeyDimensionColumns

Scalar Function [bib].[CommaListOfBusinessKeyDimensionColumns]

Returns a comma-delimited list of all columns that are considered part of the business keys for the dimension table.

DefaultETLBatchSizeFromSettings

Scalar Function [bib].[DefaultETLBatchSizeFromSettings]

The default batch size used for ETL loading is defined in the bib.BIB_Settings table. This function returns the value specified there.

By default the value is 10000 but you can set it to any integer value. You can also edit the generated ETL stored procedures to use a custom batch size for a particular dimension

GenerateOverwriteSafetyCatchForUpdateSQL

Scalar Function [bib].[GenerateOverwriteSafetyCatchForUpdateSQL]

Generates the SQL that is executed within the ETL to ensure old records are not overwriting newer ones - provided there is a field flagged as being the record's modified date.

UpdateClauseForNonKeyStagingColumns

Scalar Function [bib].[UpdateClauseForNonKeyStagingColumns]

Generates the SQL for the update clause within the dimension loading stored procedure.

CommaListOfStagingColumnsWithoutETLColumns

Scalar Function [bib].[CommaListOfStagingColumnsWithoutETLColumns]

Returns a comma-delimited list of all columns in the staging table except those ones added to support ETL management. i.e., the columns listed are those from the model table that are flagged for inclusion in the DWH.