



ETL Processing with SQL

The universe is transformation; our life is what our thoughts make it.

—Marcus Aurelius

The ETL process, and projects associated with it, involve extracting vital information from outside sources, transforming data into clean, consistent, and usable data, and loading it into the data warehouse. This process is vital to the success of your BI solution. It is what makes the difference between a professional, functional data warehouse versus one that is messy, insufficient, and unusable. The ETL process is also one of the longest and most challenging steps in developing a BI solution.

In this chapter, we explain how to perform the ETL tasks required for your BI solution. Our ultimate goal in both this chapter and the next is to demonstrate a technique where you use a combination of SQL programming and SQL Server Integration Service (SSIS) to create a professional ETL project that will be a cornerstone of your BI solution. We cover common SQL programming techniques used to identify issues and provide resolutions associated with the ETL process. And we show how code for these SQL techniques can be placed into views and stored procedures to be used for ETL processing.

This chapter is a prelude to Chapter 7 where we delve into how SSIS and the SQL programming techniques learned in this chapter are combined. Let's begin now by taking a look at the overall process.

Performing the ETL Programming

Figure 6-1 outlines the typical steps involved in creating an ETL process using a combination of SQL Server and SSIS. Note that the process includes deciding between filling up a table completely with fresh data, loading it incrementally (as explained in the following section), and updating any changes from the original source.

These steps are followed by locating the data to extract and examining its contents for validity, conformity, and completeness.

When you have verified that the data available meets your needs, it is likely that you still may need to manipulate it to some degree to fit your destination tables. This manipulation can come in the form of renaming columns or converting the original data types to their destination data types. Once all of these preparations have been completed, you can load the data into your data warehouse tables and begin the process again for each data warehouse table you need to fill.

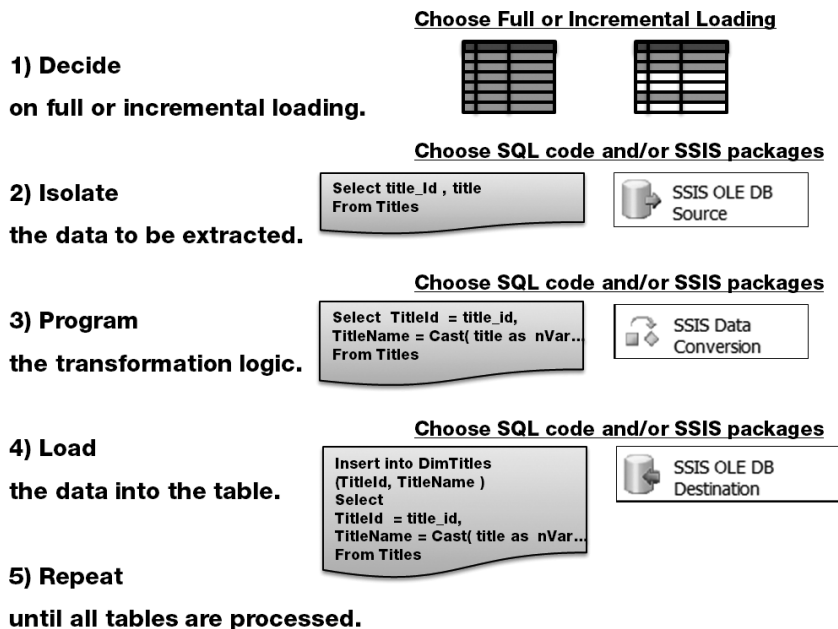


Figure 6-1. The ETL process with SQL Server and SSIS

Most of these steps can be completed using either SQL programming statements or SSIS tasks, and we examine both in this book. You will likely understand the role of the SSIS tasks more thoroughly if we start by examining the SQL statements that they represent. For that reason, let's examine the code necessary to complete these steps using SQL programming statements.

Deciding on Full or Incremental Loading

Tables in the data warehouse can be either cleared out and refilled or loaded incrementally. Clearing out the tables and then completely refilling them is known as a *flush and fill* technique. This technique is the simplest way to implement an ETL process, but it does not work well with large tables. When dimension tables are small and have only a few thousand rows, the flush and fill technique works quite well. Large tables, such as the fact table, for example, may have millions of rows. Therefore, the time it takes to completely clear the table out and then refill it with fresh data may be excessive. In those cases, filling up only data that has changed in the original source is a much more efficient choice.

To use the flush and fill technique, clear the tables in the data warehouse of a SQL Server database using either the DELETE command or the TRUNCATE command. When using the delete command, rows are deleted from the table one by one. Accordingly, if there are 1,000 rows in a table, the delete will be processed 1,000 times. This happens very quickly, but it will still take more time than a simple truncation. The TRUNCATE command de-allocates data pages that internally store the data in SQL Server. These invisible data pages are then free to be reused for other objects in a SQL Server database. Truncation represents the quickest way to clear out a SQL table, but you are not allowed to truncate a table if there are foreign key constraints associated with that table.

Listing 6-1 is an example of what your SQL code looks like using the DELETE command.

■ **Note** It may help to have the listing files open in SQL Management Studio as we discuss them. All of the SQL code we are discussing in this book is provided for you as part of the downloadable book files. The files for this chapter are found in the C:_BookFiles\Chapter06Files folder.

Listing 6-1. Deleting Data from the Data Warehouse Tables Using the Delete Command

```
Delete From dbo.FactSales
Delete From dbo.FactTitlesAuthors
Delete From dbo.DimTitles
Delete From dbo.DimPublishers
Delete From dbo.DimStores
Delete From dbo.DimAuthors
```

If you choose to use the truncation statement here, your code must include statements that drop the foreign key relationships before truncation. Listing 6-2 is an example of what your SQL code looks like using the TRUNCATE command.

Listing 6-2. Truncating the Table Data and Resetting the Identity Values

```
/****** Drop Foreign Key s *****/
Alter Table [dbo].[DimTitles] Drop Constraint [FK_DimTitles_DimPublishers]
Alter Table [dbo].[FactTitlesAuthors] Drop Constraint [FK_FactTitlesAuthors_DimAuthors]
Alter Table [dbo].[FactTitlesAuthors] Drop Constraint [FK_FactTitlesAuthors_DimTitles]
Alter Table [dbo].[FactSales] Drop Constraint [FK_FactSales_DimStores]
Alter Table [dbo].[FactSales] Drop Constraint [FK_FactSales_DimTitles]
Go

/****** Clear all tables and reset their Identity Auto Number *****/
Truncate Table dbo.FactSales
Truncate Table dbo.FactTitlesAuthors
Truncate Table dbo.DimTitles
Truncate Table dbo.DimPublishers
Truncate Table dbo.DimStores
Truncate Table dbo.DimAuthors
Go

/****** Add Foreign Keys *****/
Alter Table [dbo].[DimTitles] With Check Add Constraint [FK_DimTitles_DimPublishers]
Foreign Key ([PublisherKey]) References [dbo].[DimPublishers] ([PublisherKey])

Alter Table [dbo].[FactTitlesAuthors] With Check Add Constraint [FK_FactTitlesAuthors_DimAuthors]
Foreign Key ([AuthorKey]) References [dbo].[DimAuthors] ([AuthorKey])

Alter Table [dbo].[FactTitlesAuthors] With Check Add Constraint [FK_FactTitlesAuthors_DimTitles]
Foreign Key ([TitleKey]) References [dbo].[DimTitles] ([TitleKey])

Alter Table [dbo].[FactSales] With Check Add Constraint [FK_FactSales_DimStores]
Foreign Key ([StoreKey]) References [dbo].[DimStores] ([StoreKey])

Alter Table [dbo].[FactSales] With Check Add Constraint [FK_FactSales_DimTitles]
Foreign Key ([TitleKey]) References [dbo].[DimTitles] ([TitleKey])
Go
```

An additional benefit of truncation over deletion is that if you have a table using the identity option to create integer key values, truncation will automatically reset the numbering scheme to its original value (typically 1). Deletion, on the other hand, will not reset the number; therefore, when you insert a new row, the new integer value will continue from where the previous insertions left off before deletion. Normally this is not what you want, because the numbering will no longer start from 1, which may be confusing.

Listings 6-1 and 6-2 are examples of code used in the flush and fill process. But, if you choose to do an incremental load, do not clear the tables first. Instead, compare the values between the source and destination tables. Then either add rows to the destination tables where new rows are found in the source, update rows in the destination that are changed in the source or delete rows from the destination that are removed in the source.

In the example in Listing 6-3, a Customer's OLTP table contains a flag column called RowStatus. Each time a row in this OLTP table is added, updated or marked for deletion, a flag is set indicating the operation. The flags are examined to determine which data in the Customers table needs to be synchronized in the DimCustomers table. Then an INSERT, UPDATE or DELETE takes place depending on the flag found in the table.

■ **Tip** We have included this code in the Chapter06 folder of the downloadable files if you would like to test it.

Listing 6-3. Synchronizing Values Between Tables

```
Use TEMPDB
Go
-- Step #1. Make two demo tables
Create Table Customers
( CustomerId int
, CustomerName varchar(50)
, RowStatus Char(1) check(RowStatus in ('i','u','d') ) )
Go
Create Table DimCustomers
( CustomerId int
, CustomerName varchar(50) )
Go

-- Step #2. Add some starting data
Insert into Customers (CustomerId, CustomerName, RowStatus )
Values(1, 'Bob Smith', 'i')
Go
Insert into Customers (CustomerId, CustomerName, RowStatus )
Values(2, 'Sue Jones', 'i')
Go

-- Step #3. Verify that the tables are not synchronized
Select * from Customers
Select * from DimCustomers
Go

-- Step #4 Synchronize the tables with this code
BEGIN TRANSACTION
Insert into DimCustomers
(CustomerId, CustomerName)
Select CustomerId, CustomerName
From Customers
```

```

Where RowStatus is NOT null
    AND RowStatus = 'i'
-- Synchronize Updates
Update DimCustomers
Set DimCustomers.CustomerName = Customers.CustomerName
From DimCustomers
JOIN Customers
    On DimCustomers.CustomerId = Customers.CustomerId
    AND RowStatus = 'u'
-- Synchronize Deletes
Delete DimRows
From DimCustomers as DimRows
JOIN Customers
    On DimRows.CustomerId = Customers.CustomerId
    AND RowStatus = 'd'
-- After we import data to the dim table
-- we must reset the flags to null!
Update Customers Set RowStatus = null
COMMIT TRANSACTION

-- Step #5. Test that both tables now contain the same rows
Select * from Customers
Select * from DimCustomers
Go

-- Step #6. Test the Updates and Delete options
Update Customers
Set
    CustomerName = 'Robert Smith'
, RowStatus = 'u'
Where CustomerId = 1
Go
Update Customers
Set
    CustomerName = 'deleted'
, RowStatus = 'd'
Where CustomerId = 2
Go

-- Step #7. Verify that the tables are not synchronized
Select * from Customers
Select * from DimCustomers
Go

-- Step #8. Synchronize the tables with the same code as before
BEGIN TRANSACTION
Insert into DimCustomers
(CustomerId, CustomerName)
Select CustomerId, CustomerName
From Customers
Where RowStatus is NOT null
    AND RowStatus = 'i'
-- Synchronize Updates

```

```

Update DimCustomers
  Set DimCustomers.CustomerName = Customers.CustomerName
  From DimCustomers
  JOIN Customers
    On DimCustomers.CustomerId = Customers.CustomerId
    AND RowStatus = 'u'
-- Synchronize Deletes
Delete DimRows
  From DimCustomers as DimRows
  JOIN Customers
    On DimRows.CustomerId = Customers.CustomerId
    AND RowStatus = 'd'
-- After we import data to the dim table
-- we must reset the flags to null!
Update Customers Set RowStatus = null
COMMIT TRANSACTION

-- Step #9. Test that both tables contain the same rows
Select * from Customers
Select * from DimCustomers
Go

-- Step #10. Setup an ETL process that will run the Synchronization code

```

As you can see, creating SQL code to accomplish incremental loading can be quite complex. The good news is that you will not need to do this for most tables. Many tables are too small to benefit from the incremental approach, and in those cases, you should try to keep your ETL processing as simple as possible and stick with the flush and fill technique. For example, all the tables in our three demo databases have small amounts of data; consequently, this book focuses on the flush and fill technique for all the tables.

■ **Note** The problem with the approach we just demonstrated is that the OLTP table needs to have a tracking column. Since SQL 2005, Microsoft has introduced the SQL Merge command that performs these same comparison tasks without a tracking column. We use SQL statements in Listing 6-3 as an example of an original method that will work with most database software, but remember that there is more than one way to hook a fish. Although we don't want to confuse readers by introducing multiple ways to solve the same tasks, we have created a web page detailing a number of historic and modern approaches to this task. For more information, visit www.NorthwestTech.org/ProBISolutions/ETLProcessing.

Isolating the Data to Be Extracted

We now need to examine the data needed for the ETL process. Selecting all the data from the table you are working on is a good start. You can do this by launching a query window, typing in a simple SELECT statement, and executing it to get the results. We begin the process with a statement such as the one shown in Listing 6-4.

Listing 6-4. Selecting All the Data from the Source Table

```
Select * from [Pubs].[dbo].[Titles]
```

Formatting Your Code

Often the code you use to isolate the data is later used in the ETL process. Because of this, you may want to take time to make your code look professional. One way of making your code more professional is to format the ETL code. For example, although using a star symbol to indicate all columns implicitly is acceptable practice for ad hoc queries, a better practice is to explicitly list columns individually as we have in Listing 6-5.

Listing 6-5. Explicitly Listing the Columns

```
Select
    [title_id]
  , [title]
  , [type]
  , [pub_id]
  , [price]
  , [advance]
  , [royalty]
  , [ytd_sales]
  , [notes]
  , [pubdate]
From [Pubs].[dbo].[Titles]
```

You may notice that we are using square brackets around column and table names. This is optional; however, this convention is also considered a best practice.

One additional convention is identifying a table using its full name, or at least most of it. The standard parts of a table's name are < ServerName > . < DatabaseName > . < SchemaName > . < TableName > . It is common to use the last three parts of the fully qualified name, but you seldom use the server name part. Doing so indicates that you want to access a table on a remote server. Although this can be advantageous, it necessitates that SQL Server be configured to use linked servers, something that is not commonly done on a production server. Without a linked server, including the server name as part of the full name of the table generates an error. You therefore use the three-part name most of the time.

Identifying the Transformation Logic

Identifying which transformation is needed and then programming the transformation logic is the portion of the ETL process that usually takes the longest. Let's take a look at an example to understand what is involved. In Figure 6-2, you see two tables: the Titles OLTP database table and the DimTitles OLAP data warehouse table. Let's compare the two tables:

- The DimTitles table has fewer columns.
- The column names are different between the tables.
- The data types are different on some columns.
- There is now a surrogate key that can be used for foreign key references.
- Nullability has been changed in many columns.
- Some values have been cleansed and made more readable.

titles			
	Column Name	Data Type	Allow Nulls
?	title_id	td:varchar(6)	<input type="checkbox"/>
	title	varchar(80)	<input type="checkbox"/>
	type	char(12)	<input type="checkbox"/>
	pub_id	char(4)	<input checked="" type="checkbox"/>
	price	money	<input checked="" type="checkbox"/>
	advance	money	<input checked="" type="checkbox"/>
	royalty	int	<input checked="" type="checkbox"/>
	ytd_sales	int	<input checked="" type="checkbox"/>
	notes	varchar(200)	<input checked="" type="checkbox"/>
	pubdate	datetime	<input type="checkbox"/>
			<input type="checkbox"/>

DimTitles			
	Column Name	Data Type	Allow Nulls
?	TitleKey	int	<input type="checkbox"/>
	TitleId	nvarchar(6)	<input type="checkbox"/>
	TitleName	nvarchar(50)	<input type="checkbox"/>
	TitleType	nvarchar(50)	<input type="checkbox"/>
	PublisherKey	int	<input type="checkbox"/>
	TitlePrice	decimal(18, 4)	<input type="checkbox"/>
	PublishedDate	datetime	<input type="checkbox"/>
			<input type="checkbox"/>

Figure 6-2. Comparing the Titles table to DimTitles

All these differences make up a list of transformations that must be addressed as the data moves from the Titles table to the DimTitles table.

■ **Note** The following pages have a lot of SQL programming code. If you are not a SQL programmer, many examples will seem obscure and perhaps even difficult to read. We have endeavored to keep the examples simple to alleviate confusion; however, the ETL process is complex and most examples can be simplified only so much. Consequently, consider our listings as general examples of how a programmer could create an ETL process. Not every ETL process will be coded the same way, and they may not be this simplistic. If you happen to find these samples too difficult, keep in mind that you may never be asked to create the SQL code on your own. Nevertheless, you may be expected to understand what some of this SQL code does. Therefore, we recommend that you focus on the explanation of each process instead of the details on how the code is written.

Programming Your Transformation Logic

You need to create code and programming structures to transform any data that requires it. This transformation code can use SQL or application code, such as C#, or a combination of both.

This code is generated for you using tools such as SSIS. However, automatically generated code is not efficient code, so you may have to optimize it yourself. For that matter, sometimes you even need to fix it before you can use it in production.

The more you work with ETL processing, the more you will find that having a thorough understanding of the code that performs the transformations will help you effectively create SSIS packages. Using tools that help you create code and knowing how to optimize that code are two aspects of ETL processing that go hand in hand. Let's take a look at some common programming techniques that are simple to implement but still provide a great deal of benefit for your efforts.

Reducing the Data

It is unlikely that you will need to extract every column from the original table; therefore, you can simply leave out the columns you do not want from the select clause. This simple procedure represents the first task in optimizing your ETL process.

Listing 6-6 shows code requesting only the columns needed to fill up the DimTitles table. If you compare the listed columns to the possible Titles table's columns shown in Figure 6-2, you see that this represents about a 30% reduction over selecting all the columns by using a query such as `SELECT * FROM Titles`.

Listing 6-6. Selecting Only Data Required for the Destination Table

```
Select
    [title_id]
  , [title]
  , [type]
  , [pub_id]
  , [price]
  , [pubdate]
From [Pubs].[dbo].[Titles]
```

Using Column Aliases

You may want to change the name of your source columns to match the names of your destination columns by using a column alias. This makes it easier for others to see the correlation between your sources and destinations, as well as aids in the SSIS configuration (covered in the next chapter). SQL Server allows you to create column aliases in two formats; the first is `[column name] AS [alias]`, and the second is `[alias] = [column name]`. Both of these accomplish the same thing, and both are shown in Listing 6-7.

Listing 6-7. Using Different Styles of Column Aliases

```
-- Older style column aliases: [column name] as [alias]
Select
    [title_id] as [TitleId]
  , [title] as [TitleName]
  , [type] as [TitleType]
  --, [pub_id] Will be replaced with a PublisherKey
  , [price] as [TitlePrice]
  , [pubdate] as [PublishedDate]
From [Pubs].[dbo].[Titles]

-- Newer style column aliases: [alias] = [column name]
Select
    [TitleId] = [title_id]
  , [TitleName] = [title]
  , [TitleType] = [type]
  --, [pub_id] Will be replaced with a PublisherKey
  , [TitlePrice] = [price]
  , [PublishedDate] = [pubdate]
From [Pubs].[dbo].[Titles]
```

Tip The first style is an older one that many programmers, including the authors of this book, were introduced to when we first started SQL programming. Although it is familiar and simple, it has one basic problem: the aliases are more difficult to spot when reading the code. The newer style aligns all the aliases on the left side of the column listings and provides for easier reading and troubleshooting. Perhaps this is why Microsoft documentation typically uses the left-hand alias style. But whatever the reason may be, to provide consistency, the newer style is what we use throughout this book.

Converting the Data Types

Another transformation is data type conversion between the source and destination tables. Listing 6-8 shows SQL code that converts the data types as the data is selected from the source table. This is a very simple and fast way of performing these types of transformations.

Listing 6-8. Using the Function to Convert Data Types

```
Select
    [TitleId]=Cast( [title_id] as nvarchar(6) )
    , [TitleName]=Cast( [title] as nvarchar(50) )
    , [TitleType]=Cast( [type] as nvarchar(50) )
    --, [pub_id] Will be replaced with a PublisherKey
    , [TitlePrice]=Cast( [price] as decimal(18, 4) )
    , [PublishedDate]=[pubdate] -- has the same data type in both tables
From [Pubs].[dbo].[Titles]
```

If you have programmed in SQL before, you may have noted that this listing utilizes the CAST() function to change existing data types from the source table. The cast function is one of the oldest and simplest functions in Microsoft SQL Server. It has a sister function called CONVERT() that can also perform these conversions. The CONVERT() function has additional options that are not available in CAST(). One example of this is how the CONVERT() function can be used to change the format of the date. Figure 6-3 shows an example.

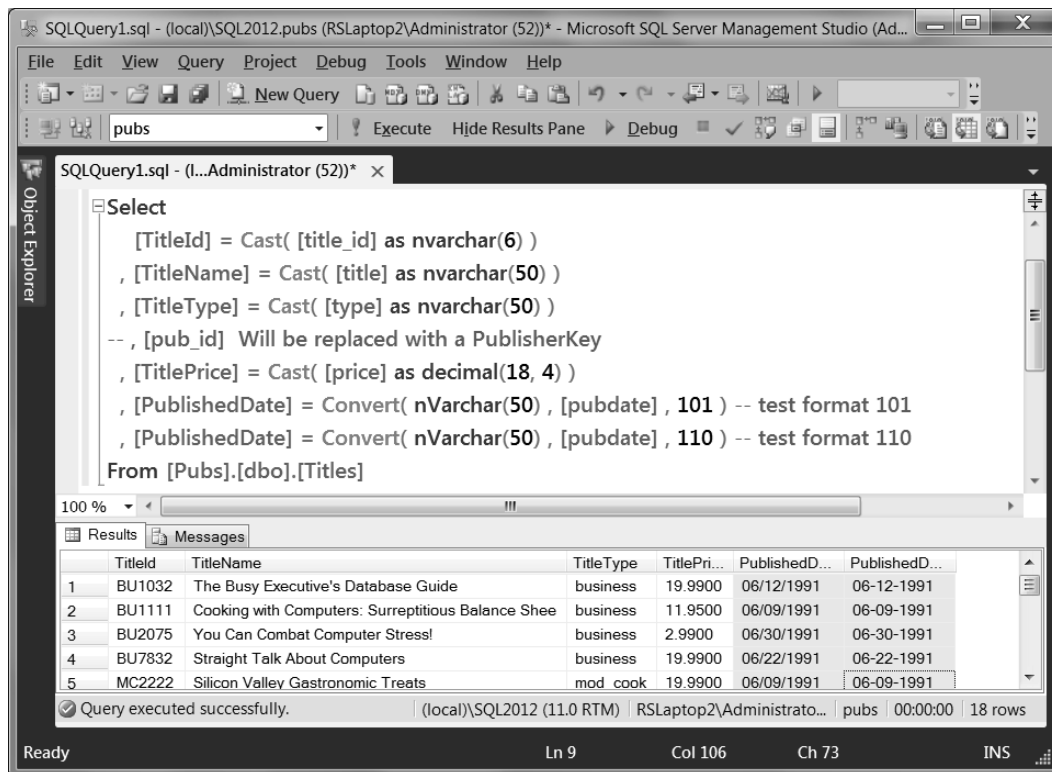


Figure 6-3. Using the CONVERT() function with additional formatting options

As you can see, using the `CONVERT()` function allows you to change datetime data types into a more readable character format, but because we are not interested in storing the dates in a character format, it is irrelevant in our current ETL process. It is nice to know that it exists, however, and that it provides additional options beyond what the `CAST()` function provides. A word of caution: keep in mind that `CAST()` is a standard ANSI function and `CONVERT()` is not. As a result, if you need to pull source data from tables in an Oracle or MySQL database, `CAST()` is still the better option.

Looking Up Surrogate Key Values

In Chapter 5, we mentioned adding surrogate keys to your dimension tables as a best practice. These artificial columns allow you to merge data from different sources and record data changes more effectively. Therefore, it is no surprise that we use surrogate keys in addition to natural keys in all our dimension tables in this book.

This necessitates that the foreign key relationships between our data warehouse tables are based on the surrogate keys rather than original natural columns. These are different from the relationships defined in the source database. As a result, we need to identify a way to look up the new surrogate key value based on the natural key value in the original tables.

To help you more thoroughly understand this issue, take a look at Figure 6-4. In this figure you can see that the original link between the titles and publishers table in the OLTP database was defined in the `pub_id` column. This is no longer true in the data warehouse where the link is provided based on the `PublisherKey` column.

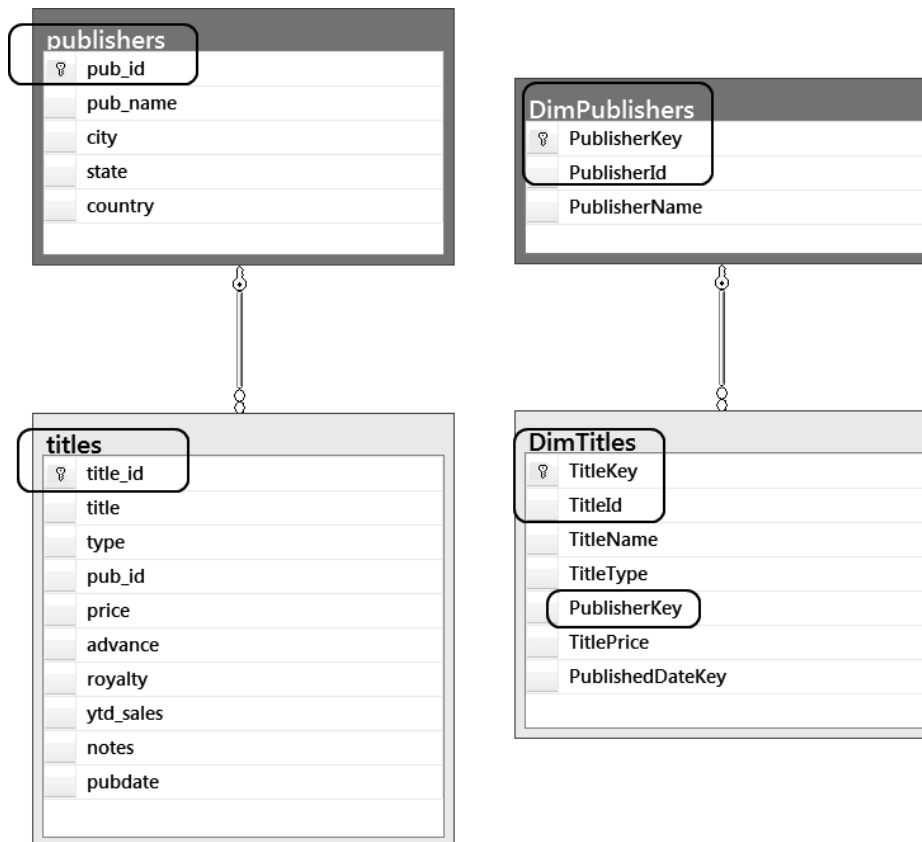


Figure 6-4. Comparing tables with and without surrogate keys

To fill the DimTitles table with correct PublisherKey data, we first need to look up the publisher's new surrogate key value by referencing the original relationship between publisher IDs that formed the original natural keys. A simple way to accomplish this is by creating a SQL join that queries the natural key columns in the source and destination tables. Listing 6-9 shows an example.

Listing 6-9. Referencing the Natural Keys to Find the Surrogate Key Value

```
Select
    [TitleId]=Cast( [title_id] as nvarchar(6) )
    , [TitleName]=Cast( [title] as nvarchar(50) )
    , [TitleType]=Cast( [type] as nvarchar(50) )
    , [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
    , [TitlePrice]=Cast( [price] as decimal(18, 4) )
    , [PublishedDate]=[pubdate]
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
    On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]
```

For the publisher key data to be available, you have to fill the DimPublishers table first. Otherwise, the surrogate key that is autogenerated will not be in existence. In Listing 6-10 we have provided a simple insert statement that accomplishes this.

Listing 6-10. Inserting Values into the DimPublishers Table

```
Insert Into [DWPubsSales].[dbo].[DimPublishers]
( [PublisherId], [PublisherName] )
Select
    [PublisherId]=Cast( [pub_id] as nchar(4) )
    , [PublisherName]=Cast( [pub_name] as nvarchar(50) )
From [pubs].[dbo].[publishers]
```

Provide Conformity

One additional common ETL task is the process of conforming data to be more readable or more consistent. There are two simple ways to accomplish this using SQL code; the first is to use a SELECT-CASE statement, and the second involves using a lookup table.

A SELECT-CASE statement is created by adding the CASE clause to any statement. For example, in Listing 6-11, we extract the values of the TitleType column in the WHEN clause and match the patterns of “business,” “mod_cook,” “popular_comp” and so on. We do this because the current values do not read well. They should have been capitalized and fully spelled out to read as “Modern Cooking” and “Popular Computing.” Because the abbreviations are not particularly legible, a transformation that conforms the data into a more readable format is in order.

To do this, examine a particular expression and then compare it to the pattern that you want to match. When the result of the expression, defined in the WHEN clause, matches a particular pattern, the output of the select statement is transformed into the value of the expression found in the THEN clause. Listing 6-11 shows how to accomplish this type of transformation using the SQL SELECT-CASE statement.

Listing 6-11. Conforming Values with a SELECT-CASE Statement

```
Select
    [TitleId]=Cast( [title_id] as nvarchar(6) )
    , [TitleName]=Cast( [title] as nvarchar(50) )
```

```

, [TitleType]=Case Cast( [type] as nvarchar(50) )
  When 'business' Then 'Business'
  When 'mod_cook' Then 'Modern Cooking'
  When 'popular_comp' Then 'Popular Computing'
  When 'psychology' Then 'Psychology'
  When 'trad_cook' Then 'Traditional Cooking'
  When 'UNDECIDED' Then 'Undecided'
End
, [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
, [TitlePrice]=Cast( [price] as decimal(18, 4) )
, [PublishedDate]=[pubdate]
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
  On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]

```

When this code runs, values such as “mod_cook” are converted into the value of “Modern Cooking,” and so on. The select statement is applied to every row in the table and takes place for every value in the column.

This method is simple and effective, but there is a downside. If you need to apply the same transformation to other tables, you need to repeat the same SQL code for each table. And if one select statement uses capitalized values, but another uses lowercase values, your tables will have multiple versions of the same data (i.e., Modern Cooking vs. modern cooking). These title type examples are unlikely to be used in any other dimension table, so it is very unlikely that this would become a problem, but it is something to keep in mind with naming conventions.

If you are going to use the same transformation data for more than one table, you may want to use the second option. The second option provides these types of transformations by creating a lookup table and then comparing the values in the lookup table to the values of the original table. When the match is found, extract the conformed value from the lookup table, replacing the original value.

The advantage of this option is that it can be reused multiple times without repeatedly having to define the list of conformed values. For example, this method is appropriate when you have a lookup table that holds two-letter state abbreviations in some table cells and the state’s full name in others. Any time you want to convert the two-letter abbreviation to the state’s full name in one or more tables, you could reference a lookup table to accomplish that goal. Because state names are likely to appear in many different tables within a given data warehouse, a lookup table is a better choice compared to using a Select-Case statement to conform your data.

Listing 6-12 shows a lookup table being created and then filled with original and transformed values via the lookup table.

Listing 6-12. Conforming Values with a Lookup Table

```

-- Create the lookup table
Create table [TitleTypeLookup] (
  [TitleTypeKey] int Primary Key Identity
  , [OriginalTitleType] nvarchar(50)
  , [CleanTitleType] nvarchar(50)
)

-- Add the original and transformed data
Insert into [TitleTypeLookup]
( [OriginalTitleType] , [CleanTitleType] )
Select
  [OriginalTitleType]=[Type]
  , [CleanTitleType]=Case Cast( [type] as nvarchar(50) )
    When 'business' Then 'Business'
    When 'mod_cook' Then 'Modern Cooking'
    When 'popular_comp' Then 'Popular Computing'

```

```

        When 'psychology' Then 'Psychology'
        When 'trad_cook' Then 'Traditional Cooking'
        When 'UNDECIDED' Then 'Undecided'
    End
From [Pubs].[dbo].[Titles]
Group By [Type] -- get distinct values

-- Combine the data from the lookup table and the original table
Select
    [TitleId]=Cast( [title_id] as nvarchar(6) )
    , [TitleName]=Cast( [title] as nvarchar(50) )
    , [TitleType]=[CleanTitleType]
    , [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
    , [TitlePrice]=Cast( [price] as decimal(18, 4) )
    , [PublishedDate]=[pubdate]
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
    On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]
Join [DWPubsSales].[dbo].[TitleTypeLookup]
    On [Pubs].[dbo].[Titles].[type]=[DWPubsSales].[dbo].[TitleTypeLookup].[OriginalTitleType]

```

■ **Note** Rather than using the `DISTINCT` keyword, you may find it advantageous to use a `SQL GROUP BY` statement that can give you different results than the `DISTINCT` keyword will when working with some functions, such as `ROW_NUMBER()`. In the example in Listing 6-12, there is no change in results, but it is a good technique to keep in mind as part of your ETL toolkit.

Generate Date Data

There are times when the ETL process will not just copy and transform existing data but instead will generate entirely new data. For example, many data warehouses have a table holding a sequential list of dates which are generated using an `INSERT` statement nested inside a programmatic loop.

In Chapter 5, we created a `DimDates` table, but did not fill it with data. As we will see in Chapter 8, Exercise 8-1, the SQL code in Listing 6-13 can be used during the ETL process to fill the `DimDates` table.

Listing 6-13. Filling the `DimDates` Table

```

-- Because the date table has no associated source table we can fill the data
-- using a SQL script.

-- Create variables to hold the start and end date
DECLARE @StartDate datetime='01/01/1990'
DECLARE @EndDate datetime='12/31/1995'

-- Use a while loop to add dates to the table
DECLARE @DateInProcess datetime
SET @DateInProcess=@StartDate

WHILE @DateInProcess<= @EndDate
BEGIN
    -- Add a row into the date dimension table for this date
    INSERT INTO DimDates (

```

```

    [Date]
, [DateName]
, [Month]
, [MonthName]
, [Quarter]
, [QuarterName]
, [Year]
, [YearName]
)
VALUES (
-- [Date]
    @DateInProcess
-- [DateName]
    , Convert(varchar(50), @DateInProcess, 110)+' , '
      + DateName( weekday, @DateInProcess )
-- [Month]
, Month( @DateInProcess )
-- [MonthName]
, Cast( Year(@DateInProcess) as varchar(4) )+' - '
  + DateName( month, @DateInProcess )
-- [Quarter]
, DateName( quarter, @DateInProcess )
-- [QuarterName]
, Cast( Year(@DateInProcess) as varchar(4) )+' - '
  + 'Q'+DateName( quarter, @DateInProcess )
-- [Year]
, Year(@DateInProcess)
-- [YearName]
, Cast( Year(@DateInProcess) as Char(4) )
)

-- Add a day and loop again
SET @DateInProcess=DateAdd(d, 1, @DateInProcess)
END

-- Check the table SELECT Top 10 * FROM DimDates

```

Note that we used SQL variables to indicate the range of our dates as starting on 1/1/1990 and ending on 12/31/1995. This may seem odd, but if you look at the dates in the Pubs database, you will see that all of the dates are within this range.

Dealing with Nulls

Null values represent a special challenge because there are many different opinions on how to handle them in a data warehouse environment. Because of their very nature, nulls present you with multiple options, and it's up to you to sort out which approach works best in a given case. Nulls are most often considered to be an unknown value, but what does the term *unknown* really mean? Does it mean that it is unknowable and could never be known? Does it mean that is unknown at the moment but will soon be available? Does it mean that a value just does not apply in this particular instance? Any of these can be true: the value may not be known, may be missing, or may not be applicable.

Because of the ambiguity of the term *null*, a decision has to be made as to which interpretation is most accurate, or you may face endless arguments over the validity of your reports. Let's break each option down to help with this decision process.

Nulls in a Fact Table

Null values in the fact table can be either dimensional keys or measured values. When the null is a measured value, we recommend that you keep the null value indicating that it is either unknown or simply does not exist yet. This is because most database products can properly aggregate null values into totals and subtotals. If you substitute values such as a zero for null, the calculations may become incorrect.

To understand this issue, take a look at Listing 6-14. It shows an example of the difference when using the aggregate function `AVERAGE()`.

Listing 6-14. How Nulls Work with Aggregate Functions

```
-- Using a null you get the correct answer of 4
Select Avg([Amt]) From (
  Select [Amt] = 2
  Union
  Select [Amt] = 6
  Union
  Select [Amt] = null
) as aDemoTableWithNull

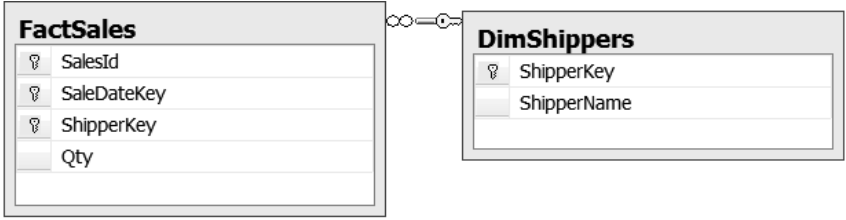
-- Using a zero you get the incorrect answer of 2
Select Avg([Amt]) From (
  Select [Amt] = 2
  Union
  Select [Amt] = 6
  Union
  Select [Amt] = 0
) as aDemoTableWithZero
```

In cases where the fact table's null value is a dimensional key, we suggest that you create an artificial key value that can further describe the meaning of the null.

Be aware that you may need to provide more than one artificial key for each interpretation of the null values. Programmers often want to group null data together and assign implicit meanings to these nulls. Although this approach is useful for determining how many orders have not shipped, there is a downside to this grouping. What happens when the implicit meaning is not correct? For example, if a package was shipped but the shipping date was not recorded, the package may be sent twice.

To alleviate these issues, provide ways to replace null values with a set of descriptive dimensional attributes. You can then use a `SELECT-CASE` lookup transformation, similar to the one we just discussed in the "Provide Conformity" section of this chapter, to exchange the null values for these more descriptive dimensional values.

Here is an example of a simple null lookup: if you have a sales record in your fact table that does not have a `ShipperId` associated with it, for no other reason than one has not been chosen yet, you could use a `ShipperId` whose name column had a value of "Shipper not selected," as shown in Figure 6-5.



RSLAPTOP2\SQL200... - dbo.FactSales

	SalesId	SaleDateKey	ShipperKey	Qty
	100	456	2	34
	101	456	-1	50
*	NULL	NULL	NULL	NULL

2 of 2

RSLAPTOP2\SQL20...dbo.DimShippers

	ShipperKey	ShipperName
	-1	Shipper not seleted
	1	FedEx
	2	UPS
	3	USPS
*	NULL	NULL

Figure 6-5. Using a surrogate key with null values

Note We use negative numbers for the null related dimensional keys, because SSAS does not handle using a zero as a dimension key well. But even if it did, the advantage to using negative numbers is that you can specify multiple interpretations by marking each with a different number such as -1, -2, -3, and so on, and easily distinguish them from the shippers that have positive ID numbers.

Nulls in a Dimension Table

We do not recommend leaving null values in dimensional tables. Instead, they can be either excluded or transformed.

The simplest technique to handle null values is to disallow them in the data warehouse. The idea is that these null values can still be reported against the OLTP environment when necessary but are too ambiguous to be used in the data warehouse. For specific occasions where disallowing nulls is appropriate, it is easily handled, as shown in Listing 6-15.

Listing 6-15. Excluding Nulls with the Where Clause

```
Select
    [TitleId]=Cast( [title_id] as nvarchar(6) )
    , [TitleName]=Cast( [title] as nvarchar(50) )
    , [TitleType]=Cast( [type] as nvarchar(50) )
```

```

    , [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
    , [TitlePrice]=Cast( [price] as decimal(18, 4) )
    , [PublishedDate]=[pubdate]
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
    On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]
Where [Pubs].[dbo].[Titles].[Title_Id] Is Not Null
And [Pubs].[dbo].[Titles].[Title] Is Not Null
And [Pubs].[dbo].[Titles].[Type] Is Not Null
And [Pubs].[dbo].[Titles].[Price] Is Not Null
And [Pubs].[dbo].[Titles].[PubDate] Is Not Null

```

This approach works for some tables; however, it has the potential to exclude much of the data that you need for your BI solution. Whenever possible, it is more accurate to transform nulls into descriptive values. One way to accomplish this is by using the ISNULL() function. In Listing 6-16 the ISNULL() function converts null values into the word *Unknown*. The value of Unknown is more descriptive than the use of null, although admittedly not by much.

Listing 6-16. Converting Null Values with the ISNULL() Function

```

Select
    [TitleId]=Cast( isNull( [title_id], -1 ) as nvarchar(6) )
    , [TitleName]=Cast( isNull( [title], 'Unknown' ) as nvarchar(50) )
    , [TitleType]=Cast( isNull( [type], 'Unknown' ) as nvarchar(50) )
    , [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
    , [TitlePrice]=Cast( isNull( [price], -1 ) as decimal(18, 4) )
    , [PublishedDate]=isNull( [pubdate], '01/01/1900' )
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
    On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]

```

Nulls in columns that consist of noncharacter data types, such as integers and datetime, will not accept the string value of Unknown. Consequently, you can use the ISNULL() function to convert the published date into something like 01/01/1900. The year 1900 is a starting point for many date data types and is unlikely to mark a real event, at least in this particular database. This means that it has no relevance to an actual publication date and can be used as an indicator for missing data. Keep in mind, this date will not be appropriate when the data includes this particular date as a legitimate value.

A Null Lookup Table

Although the previous techniques for dealing with nulls are used in many data warehouses, another option is to reference lookup values in either a dedicated lookup table or existing dimensional tables.

Lookup tables (also known as *domain tables*) are additional tables that are not part of the dimension model but instead have descriptive values that can be referenced from the dimensional tables. Figure 6-6 shows the contents of a typical lookup table.

SQLQuery1.sql - (local)\...\Adm... (52)* RSLAPTOP2\SQL20...ales - Diagram_0

```

1  SELECT
2      [NullLookupId]
3      , [NullDescription]
4  FROM [DWPubsSales].[dbo].[LookupNullDescriptions]

```

Results Messages

	NullLookupId	NullDescription
1	1	Unknown
2	2	Data Corrupted
3	3	Not Applicable
4	4	Unavailable

Figure 6-6. A typical lookup table

For the lookup table to function properly, you have to determine the meaning of each null found in the source data and transform the null value into a lookup ID that will cross-reference the lookup table.

The dilemma with a dedicated lookup table is that each dimension with null values now forms a snowflake design. Although this is not a problem per se, it violates the “keep it simple” rule.

Often you will find that a lookup table can be collapsed into your dimension tables. For example, when a date column in the fact or dimension table has a null value, you can use a dimension table such as the DimDates table as a lookup table, as long as it includes rows that contain lookup data. These lookup rows contain descriptive values that represent your null interpretations. Listing 6-17 shows an example of adding two lookup rows to the DimDates table.

Listing 6-17. Adding Additional Lookup Values to the DimDates Table

```

Set Identity_Insert [DimDates] On
INSERT INTO [DWPubsSales].[dbo].[DimDates] (
    [DateKey] -- This is normally added automatically
    , [Date]
    , [DateName]
    , [Month]
    , [MonthName]
    , [Quarter]
    , [QuarterName]
    , [Year]
    , [YearName]
)
VALUES
(
    -1 -- This will be the Primary key for the first lookup value
    , '01/01/1900'
    , 'Unknown Day'
    , -1
    , 'Unknown Month'
    , -1
    , 'Unknown Quarter'
    , -1

```

```

, 'Unknown Year'
)
, -- add a second row
(-2 -- This will be the Primary key for the second lookup value
, '02/01/1900'
, 'Corrupt Day'
, -2
, 'Corrupt Month'
, -2
, 'Corrupt Quarter'
, -2
, 'Corrupt Year'
)
Set Identity_Insert [DimDates] Off

```

Note We now have lookups for two interpretations of null values, unknown and corrupt. Remember that you can create as many null definitions as you want, but you will also have to create additional programming logic to distinguish between each case. Determining the meaning of a null value is not an easy task, nor is it within the scope of this book. A temporary simplistic approach is to use one interpretation for all your nulls, such as Unknown, until you have time to create a programmatic resolution.

In Chapter 5, you created the DimDates table with an IDENTITY option on the DateKey column. This option automatically inserts numeric values every time a new row is added. In Listing 6-13 we added rows of dates to the DimDates table but did not include null lookup values.

By default the IDENTITY option prevents values from being inserted into the column manually. However, you can manually insert a value if you enable SQL's IDENTITY_INSERT option, by using the SET IDENTITY_INSERT <table name> ON SQL command. Once that is done, additional lookup values can be added to the dimension table, as shown in Figure 6-7.

	DateK...	Date	DateName	M...	MonthName	Q...	QuarterName	Y...	YearName
1	-2	1900-02-01 ...	Corrupt Day	-2	Corrupt Month	-2	Corrupt Quarter	-2	Corrupt Year
2	-1	1900-01-01 ...	Unknown Day	-1	Unknown Month	-1	Unknown Quarter	-1	Unknown Year
3	1	1990-01-01 ...	01-01-1990, ...	1	1990 - January	1	1990 - Q1	1...	1990
4	2	1990-01-02 ...	01-02-1990, ...	1	1990 - January	1	1990 - Q1	1...	1990
5	3	1990-01-03 ...	01-03-1990	1	1990 - January	1	1990 - Q1	1	1990

Figure 6-7. Additional lookup dates have been added.

Once you add the new lookup rows, you can use them in your other tables. Listing 6-18 shows code that will do this very thing. It utilizes an outer join to include every value in the DimTitles, even when no matching date is found in the DimDates table. On every row where a matching date is not found, the outer join forces a null value

to automatically be inserted into the results. Therefore, we can use the `ISNULL()` function to convert the new null value into `-1`, which we then cross-reference to the Unknown date in the `DimDates` table.

Listing 6-18. Cross-Referencing the `DimDates` Table's Dimensional Keys

```
Select
    [TitleId]=Cast( isNull( [title_id], -1 ) as nvarchar(6) )
    , [TitleName]=Cast( isNull( [title], 'Unknown' ) as nvarchar(50) )
    , [TitleType]=Cast( isNull( [type], 'Unknown' ) as nvarchar(50) )
    , [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
    , [TitlePrice]=Cast( isNull( [price], -1 ) as decimal(18, 4) )
    , [PublishedDateKey]=isNull( [DWPubsSales].[dbo].[DimDates].[DateKey], -1 )
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
    On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]
Left Join [DWPubsSales].[dbo].[DimDates] -- The "Left" keeps dates not found in DimDates
    On [Pubs].[dbo].[Titles].[pubdate]=[DWPubsSales].[dbo].[DimDates].[Date]
```

■ **Note** The problem with each of these solutions is your ability to interpret the meaning of each null value. If your source data does not provide you any way to identify the meaning, the null values will have to be assigned as an Unknown. This is not particularly satisfying, but there is no magic that can fix these issues. You must work to improve the source data before you can resolve the interpretation of nulls in your data warehouse.

The SQL Query Designer

As you can see, SQL code can become complex very quickly. To make the programming process easier, you can utilize the SQL Query Designer that comes as part of SQL Management Studio or the one that comes with Integration Services (SSIS). Both these tools are nearly identical and create SQL code using a GUI interface.

The Query Designer that is part of SQL Management Studio makes it easy for you to create multiple statements in a single SQL script file, whereas the one that is part of the SSIS development environment does not. Therefore, it is likely that you will find Management Studio's Query Designer to be the more powerful of the two options.

To start SQL Server Management Studio's Query Designer, open a new query window and simultaneously press the `Ctrl+Shift+Q` keys. You can also access the designer through the Query menu at the top of Management Studio or by right-clicking a blank area of a query window and selecting the Design Query in Editor option from the context menu. We recommend using this last option. Usually, we start by adding a comment in the query window to identify what we are trying to accomplish, then creating a new line below the comment and right-clicking the blank spot to access the context menu (Figure 6-8).

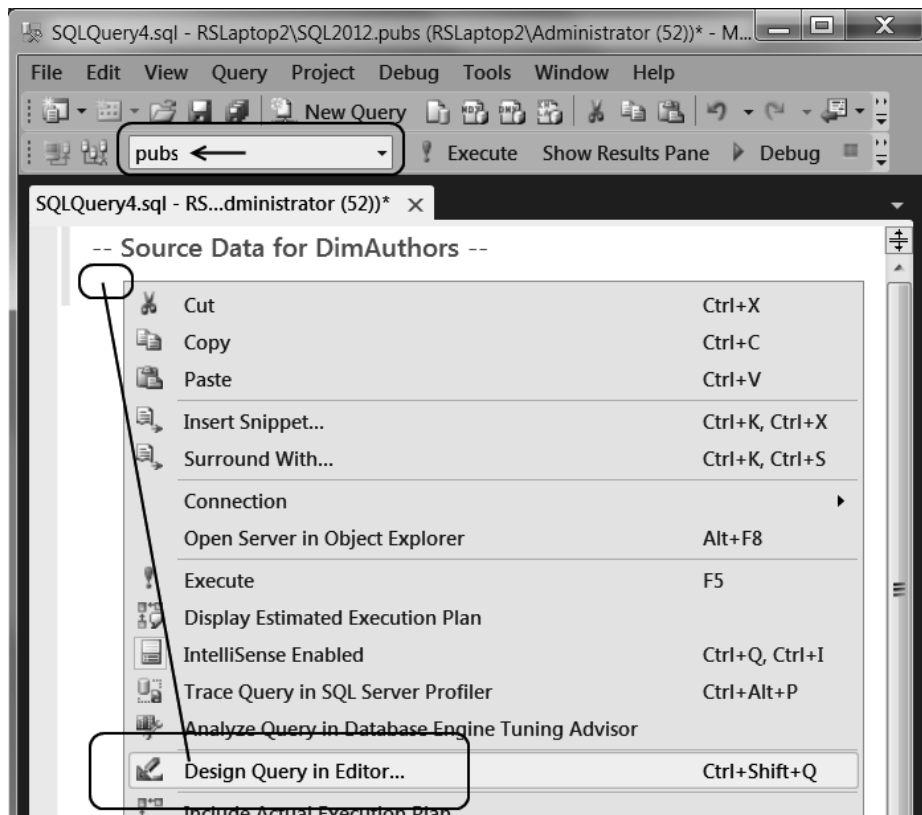


Figure 6-8. Starting the query editor

When the Query Designer launches, it will present the Add Table dialog window and a list of database tables. Selecting one or more tables in this window adds the table names to the FROM clause of the new SQL SELECT statement you are creating. For example, if you select the Authors table and click the Add button, the Authors table will be added to the query window (Figure 6-9).

■ Important When the Query Designer is open, you cannot choose which database to use. If you do not see the tables you were expecting, close the Query Designer, change the focus to the correct database, and then reopen the Query Designer. You can use the database selector, indicated with a circle and an arrow in Figure 6-8, to change the database focus.

If you want your query to include more than one table, you can hold down the Ctrl button, select multiple tables, and then click the Add button. This feature allows you to create complex queries, such as SQL joins. After you have selected all the tables you want to use, click the Close button to close the Add Tables dialog window.

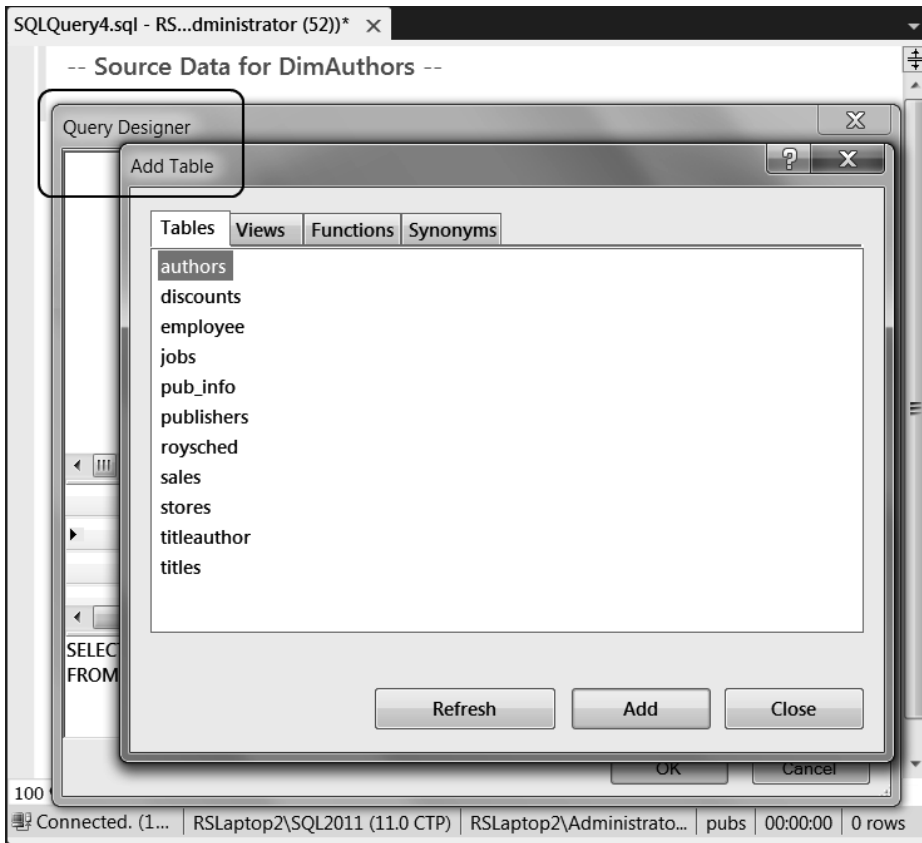


Figure 6-9. Selecting the tables in the Query Designer

When the Add Tables dialog window closes, your tables will be represented at the top of the Query Designer window, and the SQL code to query the table will be at the bottom (Figure 6-10).

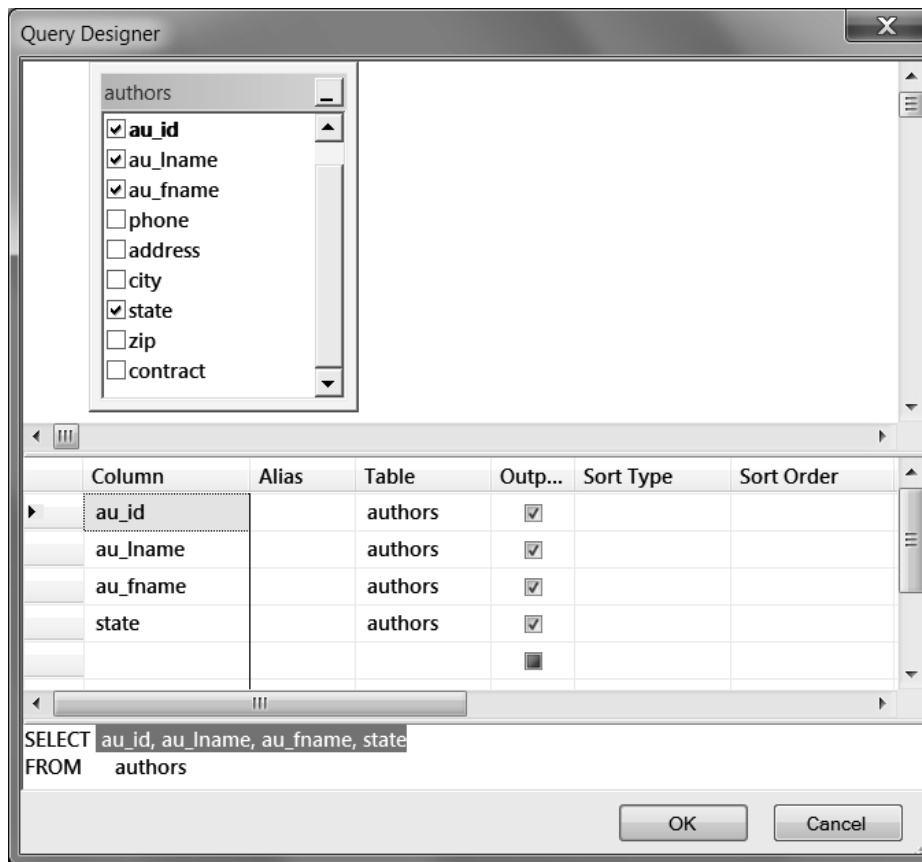


Figure 6-10. Selecting the columns in the Query Designer

We need to extract data from the original Author's source table. Therefore, we write a SQL SELECT statement to retrieve this data. If you check the checkboxes on each column you want to use, the Query Designer will add them to the SELECT statement you are building (Figure 6-10).

The Authors and DimAuthors tables contain several differences. As a result, we need to apply a number of transformations to the basic query. These include combining data from multiple columns into one column of data, renaming the existing columns, and converting column data types.

Common transformations are concatenating two columns to form a single name and converting source data into a different data type. In our example, we want to combine an author's first and last name into a single column. Keeping the names separate in an OLTP database is common, but it is unlikely to be of use in our sales data warehouse, that is, unless you think that reports will be made that will aggregate the measures on a person's first or last name.

In the DWPubsSales data warehouse, we are going to assume that a sales report detailing the sales quantity based on an author's last name is not needed. Therefore, we are going to combine an authors' first and last names to keep the reporting process as simple as possible. To combine this data, we can add a simple concatenation to the query, as shown in Figure 6-11.

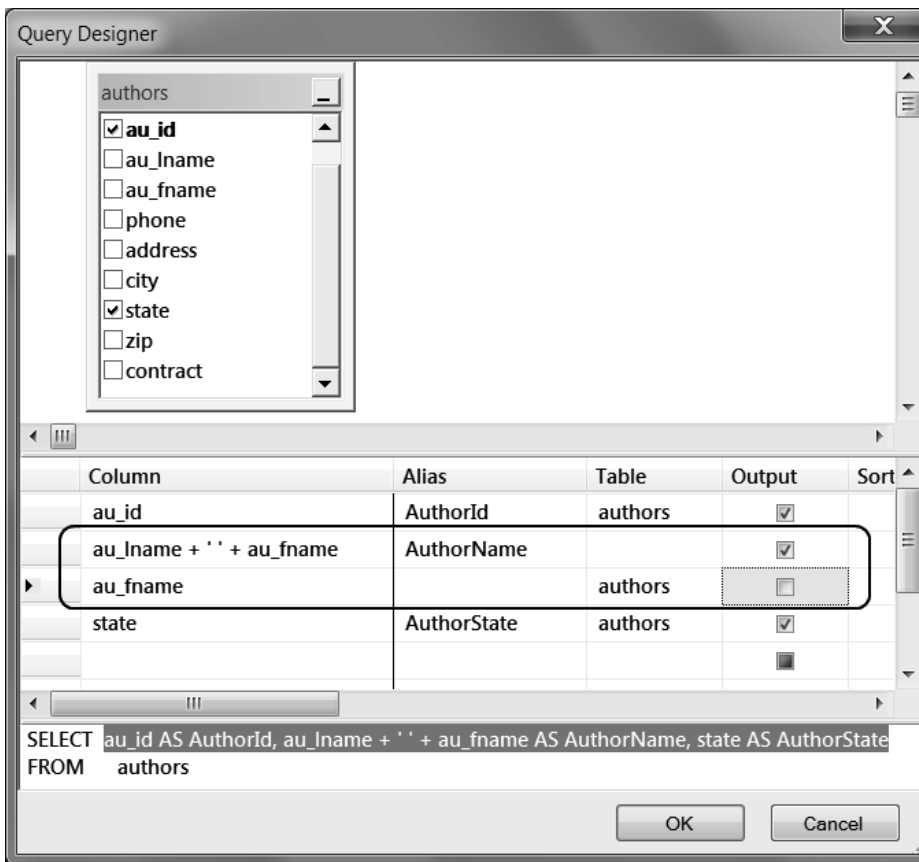


Figure 6-11. Using SQL expressions in the Query Designer

Renaming columns is about the simplest transformation you can do. This transformation is accomplished by adding the alias to the Alias column in the Query Designer (Figure 6-11).

As we have seen previously, you change data types with either the `CAST()` or `CONVERT()` function, and both of these can be typed right into the query window to form another SQL expression.

There is no reason why you cannot combine both the concatenation and added casting transformations as shown in Figure 6-12. Unfortunately, the Query Designer tool is not designed for this and will misunderstand what you are trying to do even though you are using perfectly legal SQL syntax. As you click away from the Column cell where your expression is typed, you will receive an error message stating that the conversion may be unnecessary (Figure 6-12). You can ignore this error message as long as you are sure that the SQL syntax is correct and appropriate to what you are trying to accomplish, but at least it will change your code, in spite of having to work around this error message.

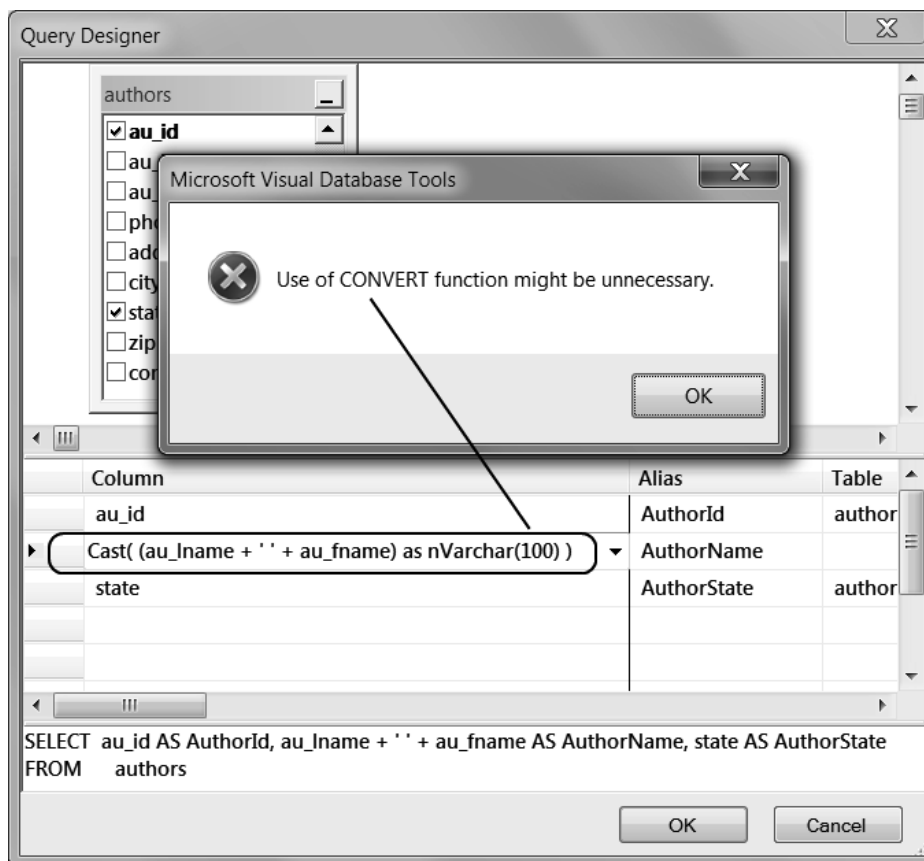


Figure 6-12. A misleading error from the Query Designer

Your formatted code will now look something like the code in Listing 6-19. Notice that an N has been added to the data conversion. This N indicates that the space between the two single quotes should be considered Unicode. Strictly speaking, there is nothing wrong with this; however, it may be confusing to someone working with this code. Using the N symbol is well documented and can easily be researched to determine the meaning, but it is still commonly misunderstood by developers new to SQL programming.

Listing 6-19. An Example of Code That the Query Designer Will Change Without Your Consent

```
SELECT au_id AS AuthorId, CAST( (au_fname+N' ' +au_lname) AS nVarchar(100)) AS AuthorName, state
AS AuthorState
FROM authors
```

Another issue with the Query Designer is the way it formats your code. As we mentioned at the beginning of this chapter, formatting your code consistently is the mark of a professional. If you try to format the code in the Query Designer window, however, it will ignore what you type and change the code to what it thinks is better. Yes, the Query Designer does have an ego.

You can always clean up the code and the conversion code after you close the Query Designer, but if you forget this feature, it can be frustrating. Figure 6-13 shows how we reformatted the code created using the Query Designer. We have also added a data conversion to the state column after the Query Designer closes to avoid seeing the error in Figure 6-12 again.

Note To solve the issues of code conversion and reformatting, we recommend using the Query Designer to type out the basic syntax for your SQL statements and then closing the tool and completing the statement by hand. Using this combination will provide you with a quick and easy way to write SQL code.

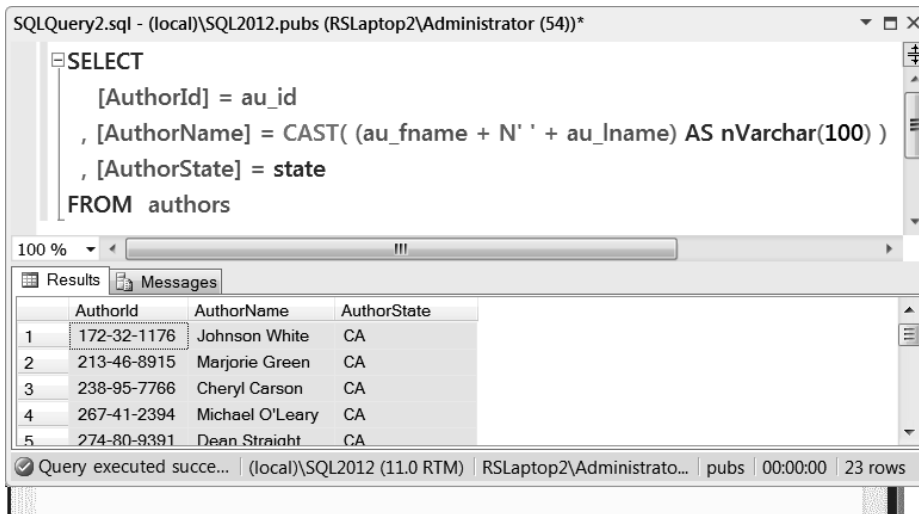


Figure 6-13. Reformatting and updating your query code

Updating Your BI Documentation

This chapter has focused on developing SQL code for your ETL process. The general consensus seems to be that most developers dislike creating documentation. And yet, documentation is an important aspect of completing any solution, and that documentation must be upgraded as progress is made toward the completion of the solution.

In this book, we have tried to keep the documentation requirements to a minimum. So far, we have simplified this process by using an Excel spreadsheet to document the solution objects. Because this is currently our only form of documentation, we need to keep it up-to-date by recording our transformation logic in the spreadsheet.

Figure 6-14 shows an example of adding the transformations required to the data warehouse objects worksheet. Doing this provides two distinct advantages. First, it gives you the opportunity to let other developers know what kind of transformations have occurred without having to read complex SQL statements. And second, it provides you with a way of verifying that you have completed coding all the transformations in your list.

Pubs.dbo.titles.pubdate	datetime	int	lookup in Dim table
Pubs.dbo.publishers	Table	Table	
Generated	na	int	na
Pubs.dbo.publishers.pub_id	char(4)	nchar(4)	Cast to nchar(4)
Pubs.dbo.publishers.pub_name	varchar(40)	nvarchar(50)	Cast to nvarchar(50)
Pubs.dbo.titleauthor	Table	Table	
Pubs.dbo.titleauthor.title_id	varchar(6)	int	na
Pubs.dbo.titleauthor.au_id	varchar(11)	int	na
Pubs.dbo.titleauthor.au_ord	tinyint	int	Cast to Int
Pubs.dbo.authors	Table	Table	
Generated	na	int	na
Pubs.dbo.authors.au_id	varchar(11)	nchar(11)	Cast to nvarchar(50)
Pubs.dbo.authors.au_fname	varchar(40)	nvarchar(100)	Cast(Fname + Lname) to nvarchar(100)
Pubs.dbo.authors.au_lname	varchar(20)	na	na
Pubs.dbo.authors.state	char(2)	nchar(2)	Cast to nchar(2)
na	na	Table	

Figure 6-14. Documenting your ETL transformations

Note When creating your ETL code, updating your documents to define the transformations required may be the most convenient place to start. Although we do not believe that this is a replacement for formal documentation, it can allow developers to track their updates and provide you with accurate information about those changes. Later this document can be turned into formal documentation. We show you examples of this in Chapter 19.

Building an ETL Script

As you work with the ETL process, it is a good idea to organize all the ETL code you generate. This can easily be done by opening up SQL Server Management Studio to create a new query, typing in the necessary code, and saving the code as a SQL script file. The advantages to this are that you can later add script to your SSIS project as a miscellaneous file, and it can also be reused in the future for other projects that are similar. In the next exercise, you do just that.

EXERCISE 6-1. CREATING AN ETL SCRIPT

In this exercise, you create the ETL code needed to fill the Publication Industries data warehouse. You can choose either to type the SQL code presented here or to use the Query Designer tool to accomplish your goal. We recommend using both.

Important: You are practicing administrator-level tasks in this book, so you need administrator-level privileges. The easiest way to achieve this when opening a program is to remember to always right-click a menu item, select **Run as Administrator**, and then answer **Yes** to access administrator-level privileges while running this program. In Windows 7 and Vista, logging in with an administrator account is not enough. For more information, search the Web on the keywords “Windows 7 True Administrator and User Access Control.”

1. Open Excel from the Start menu using the **Run as Administrator** option.
2. Locate and open the following Excel spreadsheet:
C:_BookFiles\Chapter06Files\BISolutionWorksheets.xlsx. You can do so by clicking the **File** tab in Microsoft Excel and choosing the **Open** option from the menu.

3. We have added a column called Transformations to the spreadsheet you used in Chapter 5. It shows the transformations we need. Review the transformations defined on the data warehouse worksheet.
4. Open SQL Server Management Studio 2012. You can do so by clicking the Start button and navigating to All Programs ► Microsoft SQL Server ► SQL Server Management Studio. Right-click SQL Server Management Studio 2012 and click the Run as Administrator menu item. If the UAC message box appears asking, “Do you want the following program to make changes to this computer?” click Yes (or Continue depending upon your operating system) to accept this request.
5. When SQL Server Management Studio opens, choose to connect to the database engine by selecting this option in the Server Type dropdown box. Then click the Connect button to connect to the database engine. (For more information on connecting to your database, see Chapter 5.)
6. Decide on a method for creating the SQL Script either by clicking the New Query button and typing the code by hand, using the Query Designer, or both. Then, type the following code in Listing 6-20 to create the script.

Tip: This code can be found in the file C:_ BookFiles\Chapter06Files from the downloadable book files. We recommend that you type the following code yourself, but it is nice to know that you don’t have to.

Listing 6-20. The ELT Script for DWPubSales

```

/*****
The code in this file is used to create an ETL process for the
Publication Industries data warehouse.

IMPORTANT: You must run the "Creating the Publication Industries Data Warehouse.sql"
file before you can use this code. This file is in the Chapter05 folder.
*****/

-- Step 1) Code used to Clear tables (Will be used with SSIS Execute SQL Tasks)
Use DWPubSales

-- 1a) Drop Foreign Keys
Alter Table [dbo].[DimTitles] Drop Constraint [FK_DimTitles_DimPublishers]
Alter Table [dbo].[FactTitlesAuthors] Drop Constraint [FK_FactTitlesAuthors_DimAuthors]
Alter Table [dbo].[FactTitlesAuthors] Drop Constraint [FK_FactTitlesAuthors_DimTitles]
Alter Table [dbo].[FactSales] Drop Constraint [FK_FactSales_DimStores]
Alter Table [dbo].[FactSales] Drop Constraint [FK_FactSales_DimTitles]
Alter Table [dbo].[FactSales] Drop Constraint [FK_FactSales_DimDates]
Alter Table [dbo].[DimTitles] Drop Constraint [FK_DimTitles_DimDates]
-- You will add Foreign Keys back (At the End of the ETL Process)
Go

--1b) Clear all tables data warehouse tables and reset their Identity Auto Number
Truncate Table dbo.FactSales
Truncate Table dbo.FactTitlesAuthors
Truncate Table dbo.DimTitles
Truncate Table dbo.DimPublishers
Truncate Table dbo.DimStores
Truncate Table dbo.DimAuthors

```

```

Truncate Table dbo.DimDates
Go

-- Step 2) Code used to fill tables (Will be used with SSIS Data Flow Tasks)

-- 2a) Get source data from pubs.dbo.authors and
-- insert into DimAuthors
Select
    [AuthorId]=Cast( au_id as nChar(11) )
  , [AuthorName]=Cast( ( au_fname+' '+au_lname ) as nVarChar(100) )
  , [AuthorState]=Cast( state as nChar(2) )
From pubs.dbo.authors
Go

-- 2b) Get source data from pubs.dbo.stores and
-- insert into DimStores
Select
    [StoreId]=Cast( stor_id as nChar(4) )
  , [StoreName]=Cast( stor_name as nVarChar(50) )
From pubs.dbo.stores
Go

-- 2c) Get source data from pubs.dbo.publishers and
-- insert into DimPublishers
Select
    [PublisherId]=Cast( pub_id as nChar(4) )
  , [PublisherName]=Cast( pub_name as nVarChar(50) )
From pubs.dbo.publishers
Go

-- 2d) Create values for DimDates as needed.

-- Create variables to hold the start and end date

Declare @StartDate datetime='01/01/1990'
Declare @EndDate datetime='01/01/1995'

-- Use a while loop to add dates to the table
Declare @DateInProgress datetime
Set @DateInProgress=@StartDate

While @DateInProgress<= @EndDate
Begin
    -- Add a row into the date dimension table for this date
    Insert Into DimDates
    ( [Date], [DateName], [Month], [MonthName], [Quarter], [QuarterName], [Year],
      [YearName] )
    Values (
        @DateInProgress -- [Date]
      , DateName( weekday, @DateInProgress ) -- [DateName]
      , Month( @DateInProgress ) -- [Month]
      , DateName( month, @DateInProgress ) -- [MonthName]
      , DateName( quarter, @DateInProgress ) -- [Quarter]
      , 'Q'+DateName( quarter, @DateInProgress )+' - '
        + Cast( Year( @DateInProgress ) as nVarChar(50) ) -- [QuarterName]
      , Year( @DateInProgress )
    )

```

```

    , Cast( Year( @DateInProcess ) as nVarchar(50) ) -- [Year]
  )
-- Add a day and loop again
Set @DateInProcess=DateAdd( d, 1, @DateInProcess )
End

-- 2e) Add additional lookup values to DimDates
Set Identity_Insert [DWPubsSales].[dbo].[DimDates] On
Go

Insert Into [DWPubsSales].[dbo].[DimDates]
( [DateKey]
, [Date]
, [DateName]
, [Month]
, [MonthName]
, [Quarter]
, [QuarterName]
, [Year], [YearName] )
Select
  [DateKey]=-1
, [Date]=Cast( '01/01/1900' as nVarchar(50) )
, [DateName]=Cast( 'Unknown Day' as nVarchar(50) )
, [Month]=-1
, [MonthName]=Cast( 'Unknown Month' as nVarchar(50) )
, [Quarter]=-1
, [QuarterName]=Cast( 'Unknown Quarter' as nVarchar(50) )
, [Year]=-1
, [YearName]=Cast( 'Unknown Year' as nVarchar(50) )
Union
Select
  [DateKey]=-2
, [Date]=Cast( '01/01/1900' as nVarchar(50) )
, [DateName]=Cast( 'Corrupt Day' as nVarchar(50) )
, [Month]=-2
, [MonthName]=Cast( 'Corrupt Month' as nVarchar(50) )
, [Quarter]=-2
, [QuarterName]=Cast( 'Corrupt Quarter' as nVarchar(50) )
, [Year]=-2
, [YearName]=Cast( 'Corrupt Year' as nVarchar(50) )
Go

Set Identity_Insert [DWPubsSales].[dbo].[DimDates] Off
Go

-- 2f) Get source data from pubs.dbo.titles and
-- insert into DimTitles
Select
  [TitleId]=Cast( isNull( [title_id], -1 ) as nvarchar(6) )
, [TitleName]=Cast( isNull( [title], 'Unknown' ) as nvarchar(100) )
, [TitleType]=Case Cast( isNull( [type], 'Unknown' ) as nvarchar(50) )
  When 'business' Then N'Business'
  When 'mod_cook' Then N'Modern Cooking'

```

```

        When 'popular_comp' Then N'Popular Computing'
        When 'psychology' Then N'Psychology'
        When 'trad_cook' Then N'Traditional Cooking'
        When 'UNDECIDED' Then N'Undecided'
    End
    , [PublisherKey]=[DWPubsSales].[dbo].[DimPublishers].[PublisherKey]
    , [TitlePrice]=Cast( isNull( [price], -1 ) as decimal(18, 4) )
    , [PublishedDateKey]=isNull( [DWPubsSales].[dbo].[DimDates].[DateKey], -1 )
From [Pubs].[dbo].[Titles]
Join [DWPubsSales].[dbo].[DimPublishers]
    On [Pubs].[dbo].[Titles].[pub_id]=[DWPubsSales].[dbo].[DimPublishers].[PublisherId]
Left Join [DWPubsSales].[dbo].[DimDates] -- The "Left" keeps dates not found in DimDates
    On [Pubs].[dbo].[Titles].[pubdate]=[DWPubsSales].[dbo].[DimDates].[Date]
Go

-- 2g) Get source data from pubs.dbo.titleauthor and
-- insert into FactTitlesAuthors
Select
    [TitleKey]=DimTitles.TitleKey
--, title_id
, [AuthorKey]=DimAuthors.AuthorKey
--, au_id
, [AuthorOrder]=au_ord
From pubs.dbo.titleauthor
JOIN DWPubsSales.dbo.DimTitles
    On pubs.dbo.titleauthor.Title_id=DWPubsSales.dbo.DimTitles.TitleId
JOIN DWPubsSales.dbo.DimAuthors
    On pubs.dbo.titleauthor.Au_id=DWPubsSales.dbo.DimAuthors.AuthorId

-- 2h)Get source data from pubs.dbo.Sales and
-- insert into FactSales
Select
    [OrderNumber]=Cast( ord_num as nVarchar(50) )
, [OrderDateKey]=DateKey
--, title_id
, [TitleKey]=DimTitles.TitleKey
--, stor_id
, [StoreKey]=DimStores.StoreKey
, [SalesQuantity]=qty
From pubs.dbo.sales
JOIN DWPubsSales.dbo.DimDates
    On pubs.dbo.sales.ord_date=DWPubsSales.dbo.DimDates.date
JOIN DWPubsSales.dbo.DimTitles
    On pubs.dbo.sales.Title_id=DWPubsSales.dbo.DimTitles.TitleId
JOIN DWPubsSales.dbo.DimStores
    On pubs.dbo.sales.Stor_id=DWPubsSales.dbo.DimStores.StoreId

-- Step 3) Add Foreign Key s back (Will be used with SSIS Execute SQL Tasks)
Alter Table [dbo].[DimTitles] With Check
Add Constraint [FK_DimTitles_DimPublishers]
Foreign Key ( [PublisherKey] ) References [dbo].[DimPublishers] ( [PublisherKey] )

```



```

Alter Table [dbo].[FactTitlesAuthors] With Check
Add Constraint [FK_FactTitlesAuthors_DimAuthors]
Foreign Key ( [AuthorKey] ) References [dbo].[DimAuthors] ( [AuthorKey] )

Alter Table [dbo].[FactTitlesAuthors] With Check
Add Constraint [FK_FactTitlesAuthors_DimTitles]
Foreign Key ( [TitleKey] ) References [dbo].[DimTitles] ( [TitleKey] )

Alter Table [dbo].[FactSales] With Check
Add Constraint [FK_FactSales_DimStores]
Foreign Key ( [StoreKey] ) References [dbo].[DimStores] ( [StoreKey] )

Alter Table [dbo].[FactSales] With Check
Add Constraint [FK_FactSales_DimTitles]
Foreign Key ( [TitleKey] ) References [dbo].[DimTitles] ( [TitleKey] )

Alter Table [dbo].[FactSales] With Check
Add Constraint [FK_FactSales_DimDates]
Foreign Key ( [OrderDateKey] ) References [dbo].[DimDates] ( [DateKey] )

Alter Table [dbo].[DimTitles] With Check
Add Constraint [FK_DimTitles_DimDates]
Foreign Key ( [PublishedDateKey] ) References [dbo].[DimDates] ( [DateKey] )

```

7. When you are done, if you found any transformations that were missed, go back and finish them by adding the necessary SQL code to your script file. Once that is done, save your script file to: C:_BISolutions\PublicationsIndustries\PublicationIndustriesETLCode.sql.

In this exercise, you created the code that gathers data for the data warehouse objects. Most of these tables are still empty because we have defined only the source of the data, but we start filling them up using a combination of the SQL statements you have just created along with SSIS tasks in the next chapter. Before we do that, however, we need to look at three common ways of abstracting your SQL code: using views, stored procedures, and user-defined functions.

Working in the Abstract

It is common knowledge in the programming industry that building software by defining multiple layers can give you greater flexibility and lower maintenance costs. One of the classic ways of doing this is by providing an abstraction layer between the objects that contain data and the software that uses that data. In Microsoft SQL Server, this can be done with three simple database objects: views, stored procedures and functions.

The concept of abstraction basically means designing your software so that the underlying objects are always used indirectly. In the case of a database table, you do not directly connect your SSIS applications to the table itself, but instead utilize one of these abstraction objects.

The advantages of this design include the following:

- It can mask the complexity of programming statements by binding these statements to a named database object.
- Underlying data structures can undergo changes as a normal part of maintenance, but the abstraction objects will hide these changes.
- Permissions can be given to the abstraction objects and not to the underlying data structures, thus protecting the data structure from misuse.

- The same data structure can be represented multiple ways without having to create duplicate data structures.

You can probably think of more advantages than those we have listed, but these are enough to make one consider utilizing them in your ETL solutions. In fact, Microsoft has long recommended it as a best practice to do so.

Views

One of the most common tools used in a database is a SQL view. A SQL view consists of a named select statement that is saved internally in the database. The view is used as if it were a table. Indeed, it is sometimes called a *virtual table*. This is a misnomer, however, because it gives the impression that the table and the view are more similar than they truly are.

One of the biggest differences between a table and a view is that table data has a physical representation on your hard drive, whereas the view is just a saved select statement. Yet views still act as if they were tables, and you can select against the view exactly as you would against a table.

Although there are some restrictions about which SQL SELECT clauses are allowed in a view, they are still useful for many ETL processing tasks. For example, the code in Listing 6-21 creates a view around a SQL statement that extracts data required for a sales fact table. Note that the view transforms the column names using column aliases. It also transforms data found in multiple tables into a single logical table using SQL JOIN statements.

Listing 6-21. Creating a View for ETL Processing

Create View vEtlFactSalesData

as

Select

```
[OrderNumber]=ord_num
, [OrderDateKey]=DateKey
, [TitleKey]=DimTitles.TitleKey
, [StoreKey]=DimStores.StoreKey
, [SalesQuantity]=qty
From pubs.dbo.sales
JOIN DWPubsSales.dbo.DimDates
    On pubs.dbo.sales.ord_date=DWPubsSales.dbo.DimDates.date
JOIN DWPubsSales.dbo.DimTitles
    On pubs.dbo.sales.Title_id=DWPubsSales.dbo.DimTitles.TitleId
JOIN DWPubsSales.dbo.DimStores
    On pubs.dbo.sales.Stor_id=DWPubsSales.dbo.DimStores.StoreId
```

Using column aliases and SQL Joins in a query may be rather simple for experienced SQL programmers, but it is considered advanced by most novices. Once the view is created, however, the complexity of this query is masked by the view. You can then query the view using a simple SQL statement, such as the one in Listing 6-22, that is understood by everyone.

Listing 6-22. Querying the View

```
Select * from vEtlFactSalesData
```

■ **Note** SSIS has a data source view object that can be used in conjunction with your SSIS packages. This may make the concept of views seem redundant, but in practice, you will find that the SSIS data source views are somewhat limited since they can only be used within an SSIS project, whereas SQL views can be used by any application that connects to SQL Server.

The view is a very simple and effective tool, but it does have some disadvantages. You cannot define parameters on a view. Parameters allow you to pass in specific arguments to get back differing results. This is a useful technique when incrementally loading a data warehouse. For example, the code in Listing 6-23 filters out data that was not added on today's date. Because SQL views cannot contain parameters, this query could not be saved inside a Create View statement.

Listing 6-23. A Select Statement with a Parameter

```
Select
    [OrderNumber]=ord_num
, [OrderDateKey]=DateKey
, [TitleKey]=DimTitles.TitleKey
, [StoreKey]=DimStores.StoreKey
, [SalesQuantity]=qty
From pubs.dbo.sales
JOIN DWPubsSales.dbo.DimDates
    On pubs.dbo.sales.ord_date=DWPubsSales.dbo.DimDates.date
JOIN DWPubsSales.dbo.DimTitles
    On pubs.dbo.sales.Title_id=DWPubsSales.dbo.DimTitles.TitleId
JOIN DWPubsSales.dbo.DimStores
    On pubs.dbo.sales.Stor_id=DWPubsSales.dbo.DimStores.StoreId
Where pubs.dbo.sales.ord_date=@Today'sDate - This will not work in a View!
```

Stored Procedures

Like views, stored procedures consist of a named set of SQL statements. Unlike views, stored procedures can contain multiple statements, work with variables, and process transaction statements. Stored procedures can therefore be quite complex and contain hundreds of lines of SQL code, but they can also be as simple as a saved select statement. For example, Listing 6-24 holds the same select statement as the one in the view created in Listing 6-22. Notice that the syntax looks almost the same. The only difference is the command CREATE PROCEDURE in place of CREATE VIEW.

Listing 6-24. Creating a Stored Procedure for ETL Processing

Create Procedure pEtlFactSalesData
as

```
Select
    [OrderNumber]=ord_num
, [OrderDateKey]=DateKey
, [TitleKey]=DimTitles.TitleKey
, [StoreKey]=DimStores.StoreKey
, [SalesQuantity]=qty
From pubs.dbo.sales
```

```

JOIN DWPubsSales.dbo.DimDates
    On pubs.dbo.sales.ord_date=DWPubsSales.dbo.DimDates.date
JOIN DWPubsSales.dbo.DimTitles
    On pubs.dbo.sales.Title_id=DWPubsSales.dbo.DimTitles.TitleId
JOIN DWPubsSales.dbo.DimStores
    On pubs.dbo.sales.Stor_id=DWPubsSales.dbo.DimStores.StoreId
Go

```

Once you create a stored procedure, you run its code by executing it. The code to execute a stored procedure is in Listing 6-25.

Listing 6-25. Executing a Stored Procedure

Execute pEtlFactSalesData

If a stored procedure contains a SQL select statement, the selected results are returned to the client software. Client software includes SQL Server Management Studio, but as we show in Chapter 8, SSIS also acts as client software when it processes the stored procedure results.

As mentioned previously, one advantage of using stored procedures is their ability to process parameters. Parameters can be used to filter results and modify output or supplied transactional data. For example, you might filter a select statement specifically to return results that were associated with the current day's sales, as shown in Listing 6-26.

Listing 6-26. Altering the Stored Procedure to Use a Parameter

Alter Procedure pEtlFactSalesData

```

( @OrderDate datetime )
as
Select
    [OrderNumber]=ord_num
    , [OrderDateKey]=DateKey
    , [TitleKey]=DimTitles.TitleKey
    , [StoreKey]=DimStores.StoreKey
    , [SalesQuantity]=qty
From pubs.dbo.sales
JOIN DWPubsSales.dbo.DimDates
    On pubs.dbo.sales.ord_date=DWPubsSales.dbo.DimDates.date
JOIN DWPubsSales.dbo.DimTitles
    On pubs.dbo.sales.Title_id=DWPubsSales.dbo.DimTitles.TitleId
JOIN DWPubsSales.dbo.DimStores
    On pubs.dbo.sales.Stor_id=DWPubsSales.dbo.DimStores.StoreId
Where pubs.dbo.sales.ord_date=@OrderDate

```

Once a stored procedure is created, you can execute it by supplying the current date, using a built-in function such as the SQL Server GETDATE() function. If you were interested in incrementally loading your fact tables with only data that came in on a particular day, a stored procedure similar to this could be useful. Listing 6-27 shows an example of executing a stored procedure with a parameter.

Listing 6-27. Executing a Stored Procedure with a Parameter

```

Declare @TodaysDate datetime
Set @TodaysDate=Cast( GetDate() as datetime )
Execute pEtlFactSalesData

    @OrderDate=@TodaysDate

```

■ **Note** Whenever you are working with parameters like @OrderDate, list the name of the parameter on the left side of the assignment operator = and the argument @TodaysDate on the right side. This may seem backwards for beginner programmers who are used to reading an expression from left to right, $(1 + 1 = 2)$, but in programming the code is reversed: $(2 = 1 + 1)$.

Like all things in programming, stored procedures have both good and bad qualities. For the most part, their good qualities far outweigh their bad ones. Indeed, most programmers would be hard-pressed to find a strong negative aspect to using stored procedures in your ETL processing. But if you look hard enough, you will discover that stored procedures, when used in conjunction with SSIS packages, require special configurations: not difficult, just special. We look at how stored procedures are configured in SSIS in Chapter 7.

Stored procedures cannot be used as an expression. In other words, you cannot integrate a stored procedure call within a SQL select statement and have the stored procedure execute for each individual row. This is yet another aspect of stored procedures that is not all bad per se, but notable. One way around this issue is the use of user-defined functions (UDFs).

User-Defined Functions

Like views and stored procedures, UDFs consist of a named set of SQL statements. Unlike views or stored procedures, they can be used as an expression. For example, the GETDATE() function will evaluate into the current date much as the expression $5 + 6$ will evaluate into 11, such as in the SQL statement in Figure 6-15.

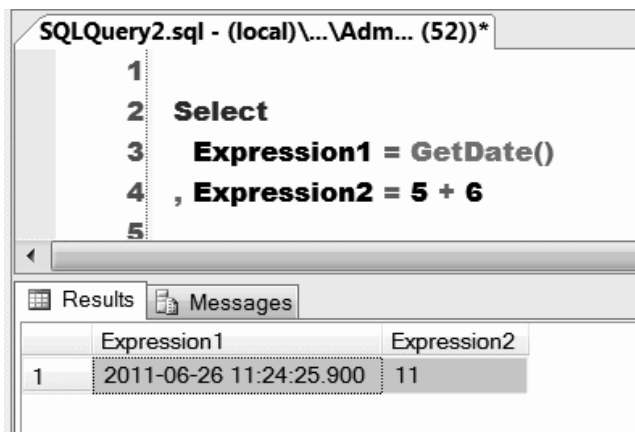


Figure 6-15. Using a function as an expression

The syntax for creating a UDF is quite similar to that of a stored procedure. First, list the name of the UDF, followed by a list of any parameters that you want to use and then the return type of the function. The return type of a stored procedure is always an implied integer, but UDFs, on the other hand, can return either a single value or a table of values. Listing 6-28 shows an example of this.

Listing 6-28. Creating a User-Defined Function

Create Function fEtlTransformStateToLongName

```
( @StateAbbreviation nChar(2) )
Returns nVarchar(50)
```

As

```
Begin
Return
( Select Case @StateAbbreviation
    When 'CA' Then 'California'
    When 'OR' Then 'Oregon'
    When 'WA' Then 'Washington'
End )
End
```

The code in Listing 6-28 creates a UDF that returns the full name of the state whenever a two-letter abbreviation for the state is given. For this example, we have added only three states, but it goes without saying that you would normally include all the states, regions, or territories that your data warehouse requires. You can test your code to verify that it works by executing select statements similar to the three test queries in Figure 6-16.

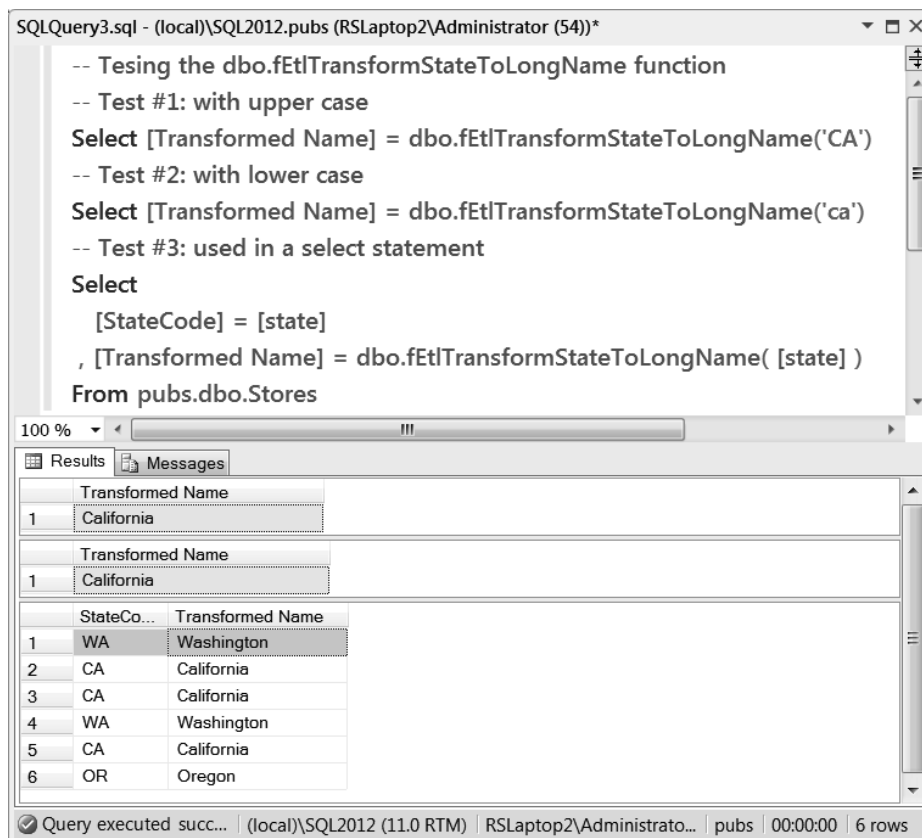


Figure 6-16. Working with user-defined functions

Moving On

We could easily devote more time to ETL programming, but we would like to take some time to demonstrate how these statements are utilized in conjunction with the SSIS. So, let's move on from here and create an SSIS project for the ETL processing needed in the DW PubsSales data warehouse. We use the SQL statements you created in this chapter to do so. Chapter 7 shows you how easy it is to combine SQL programming statements and SSIS tasks to create effective ETL projects.

LEARN BY DOING

In this “Learn by Doing” exercise, the process defined in this chapter is performed using the Northwind database. We have included an outline of the steps you performed in this chapter and an example of how the authors handled them in two Word documents. These documents are found in the folder `C:_BISolutionsBookFiles_LearnByDoing\Chapter06Files`. Please see the `ReadMe.doc` file for detailed instructions.

What's Next?

The purpose of this book is to describe the process of creating a professional BI solution in conjunction with SQL Server 2012. We have provided simple SQL code examples that allow readers to get an idea of how this code can be used as part of the ETL process. If you are a beginning SQL programmer, you may want to devote some time to improving your SQL skills. We suggest you check out a free tutorial on the W3 Schools website at <http://www.w3schools.com/sql/default.asp>. A book that we have found to be both fun and informative on SQL programming is *Beginning Microsoft SQL Server 2008 Programming* by Robert Vieira (Wrox, 2009).